Thesis for The Degree of Licentiate of Engineering

Adaptive Resource Management Techniques for High Performance Multi-Core Architectures

Nadja Ramhöj Holtryd

Division of Computer Engineering
Department of Computer Science & Engineering
Chalmers University of Technology
Gothenburg, Sweden, 2021

Adaptive Resource Management Techniques for High Performance Multi-Core Architectures

Nadja Ramhöj Holtryd

Department of Computer Science & Engineering
Division of Computer Engineering
Chalmers University of Technology and Gothenburg University
Gothenburg, Sweden

This thesis has been prepared using LaTeX.
Printed by Chalmers Reproservice,
Gothenburg, Sweden 2021.

Abstract

Reducing the average memory access time is crucial for improving the performance of applications executing on multi-core architectures. With workload consolidation this becomes increasingly challenging due to shared resource contention. Previous works has proposed techniques for partitioning of shared resources (e.g. cache and bandwidth) and prefetch throttling with the goal of mitigating contention and reducing or hiding average memory access time.

Cache partitioning in multi-core architectures is challenging due to the need to determine cache allocations with low computational overhead and the need to place the partitions in a locality-aware manner. The requirement for low computational overhead is important in order to have the capability to scale to large core counts. Previous work within multi-resource management has proposed coordinately managing a subset of the techniques: cache partitioning, bandwidth partitioning and prefetch throttling. However, coordinated management of all three techniques opens up new possible trade-offs and interactions which can be leveraged to gain better performance.

This thesis contributes with two different resource management techniques: One resource manger for scalable cache partitioning and a multi-resource management technique for coordinated management of cache partitioning, bandwidth partitioning and prefetching. The scalable resource management technique for cache partitioning uses a distributed and asynchronous cache partitioning algorithm that works together with a flexible NUCA enforcement mechanism in order to give locality-aware placement of data and support fine-grained partitions. The algorithm adapts quickly to application phase changes. The distributed nature of the algorithm together with the low computational complexity, enables the solution to be implemented in hardware and scale to large core counts. The multi-resource management technique for coordinated management of cache partitioning bandwidth partitioning and prefetching is designed using the results from our in-depth characterisation from the entire SPEC CPU2006 suite. The solution consists of three local resource management techniques that together with a coordination mechanism provides allocations which takes the inter-resource interactions and trade-offs into account.

Our evaluation shows that the distributed cache partitioning solution performs within 1% from the best known centralized solution, which cannot scale to large core counts. The solution improves performance by 9% and 16%, on average, on a 16 and 64-core multi-core architecture, respectively, compared to a shared last-level cache. The multi-resource management technique gives a performance increase of 11%, on average, over state-of-the-art and improves performance by 50% compared to the baseline 16-core multi-core without cache partitioning, bandwidth partitioning and prefetch throttling.

Keywords

Performance Isolation, Cache Partitioning, Bandwidth Partitioning, Prefetch Throttling, Multi-Core Architectures, Resource Management

Acknowledgment

I would like to thank my advisor Professor Per Stenström for his invaluable guidance. With his great knowledge in research he has explained and made complex research methodology understandable and feasible. I also would like to thank my co-advisors Associate Professor Miquel Pericàs for his enthusiasm and many ideas. I would also like to thank Dr. Madhavan Manivannan for his deep technical knowledge, insightful support and patience during my research studies. I have learned a lot from all of you.

Thanks to Fredrik Dahlgren who has been my examiner. I am also grateful for my, past and present, colleagues at Chalmers, Alexandra, Dan, Georgia, Christos, Dmitry, Sverker, Roc, Behrooz, Fatemeh, Rolf, Monica, Lars, Lena, Pedro, Jan, Johan, Evangelos, Albin, Prajith, Stefano, Petros, Stavros, Pirah, Jing and many others who created a friendly work environment.

Finally, I express my deepest gratitude to my supportive family, especially my parents, my husband Nicklas and our wonderful son Hugo.

This research has been funded by the European Research Council, under the MECCA project, contract number ERC-2013-AdG 340328, Swedish Research Council (Vetenskapsrådet) under the Approximate Algorithms and Computing Systems Project (Projekt-id: 2014-06221) and from the European Union's Horizon 2020 Programme under the LEGaTO Project (www.legato-project.eu), grant agreement no 780681. The simulations were performed on resources at Chalmers Centre for Computational Science and Engineering (C3SE) provided by the Swedish National Infrastructure for Computing (SNIC), partially funded by the Swedish Research Council through grant agreement no. 2018-05973.

List of Publications

Appended publications

This thesis is based on the following publications:

[I] N. Holtryd, M. Manivannan, P. Stenström and M. Pericàs "DELTA: Distributed Locality-Aware Cache Partitioning for Tile-based Chip Multiprocessors"
Published in IEEE International Parallel and Distributed Processing Symposium (IPDPS) 2020, New Orleans, LA, USA.

[II] N. Ramhöj Holtryd, M. Manivannan, P. Stenström and M. Pericàs "CBP: Coordinated management of cache partitioning, bandwidth partitioning and prefetch throttling"
Under review.

Contents

Chapter 1

Introduction

1.1 Background

Single core performance growth plateaued in the early 2000s, necessitating a shift towards multi-core processors [1]. This shift was mainly motivated by the end of Dennard scaling which meant that it was no longer practical to gain performance by increasing core frequency [2]. The emergence of processors with an increasing number of cores increases the off-chip memory bandwidth demand. Memory references are increasingly expensive and frequently limit processor performance. The performance of the memory system becomes a key determinant of overall system performance.

Modern multi-core processors have last level cache (LLC) banks distributed across the chip, as shown in Figure 1.1. The on-chip distances increase with additional cores, resulting in non-uniform access latencies to the cache banks. The cache banks are shared among the cores in order to maximize utilization. The off-chip memory bandwidth is also shared among the cores for the same reason. Workload consolidation is a technique to improve resource efficiency of multi-core systems by executing multiple workloads on the same physical server. However, multiple co-running applications leads to shared resource contention. Resource contention can lead to destructive interference and large performance variations across workloads, detrimentally impacting average memory access time.

In order to increase memory system performance a number of different techniques have been proposed, that aim to mitigate contention within a single shared resource. To mitigate contention, prior works have proposed partitioning of shared resources, cache [3–9] and bandwidth [10–12], with the

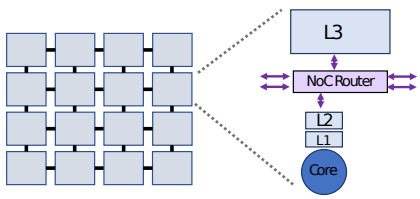


Figure 1.1: Overview of tile-based multi-core architecture.

goal of reducing average memory access time and improving performance. Prefetching [13], i.e. fetching data before it is requested to hide the memory access time, is another technique to make the memory system more efficient. However, inaccurate prefetches have been shown to increase contention [14]. Prefetch throttling [15,16], adaptively tuning when and what prefetcher settings are used based on application characteristics, has been shown to overcome the drawbacks of prefetching.

Multi-resource management is advantageous in order to avoid contention bottlenecks and exploit trade-offs between resources. It also enables coverage of more and wider range of applications compared to single resource management. Recent works have proposed combining cache and bandwidth partitioning [17–19], prefetching and cache partitioning [20, 21] and bandwidth partitioning and prefetching [22, 23] to provide additional performance gains.

This thesis proposes adaptive techniques, in the sense of reacting to workload changes, for resource management in the memory system with the goal of increasing performance of multi-core architectures.

1.2 Problem Statements

Single resource management: Prior works [7–9] have shown that in multi-core architectures locality-aware placement of data and fine-grained partitioning are key to designing a well performing cache partitioning solution. Locality-aware data placement reduces cache access times and fine-grained partitioning enables better adaptation to workloads with varying characteristics. Previous solutions have focused on solving this problem in a centralized manner which have the drawback of introducing a too high overhead when considering frequent reconfigurations for large core counts. I aim to address the following question:

Question 1: How can we design a scalable cache partitioning solution which supports locality-aware and fine-grained partitioning?

Multi-resource management: Previous works [17–23] have proposed coordinated management of only a subset of the techniques - cache partitioning, bandwidth partitioning and prefetch throttling - with the key insight that managing two instead of one is beneficial since additional trade-offs are enabled. Coordinately managing cache partitioning, bandwidth partitioning and prefetch throttling opens up new trade-offs and interactions, with significant impact on performance, which are not available when considering only a subset of the techniques. The challenge of managing multiple techniques is the increased complexity of finding a good allocation while also considering interactions among resources, which increases the computational complexity. In the context of multi-resource management I aim to answer the following question:

Question 2: How to enable coordinated management of cache partitioning, bandwidth partitioning and prefetch throttling, avoid the complexity of evaluation all possible allocations, while exploiting the new interactions and trade-offs?

1.3 Contributions

This thesis is based on two papers. In the context of single resource management Paper I answers the first question and the main contribution is:

- A fully distributed and locality-aware cache-partitioning solution which consists of a distributed allocation algorithm which asynchronously negotiates cache allocation decisions together with a reconfigurable NUCA enforcement mechanism. The distributed nature of the solution, coupled with low computational overhead, enables a hardware-based implementation. This allows the scheme to scale to large core counts while permitting frequent reconfigurations without invoking the operating system. The enforcement mechanism allows locality-aware placement of data.

In Paper II we consider multi-resource management and answer the second question. The main contributions of the paper are:

- An in-depth characterization of the performance impact of cache, bandwidth and prefetching on the entire SPEC CPU2006 suite. Our characterisation results provide several insights: i) a majority of the applications (over 90%) are sensitive to one or multiple techniques, ii) managing cache, bandwidth and prefetch opens up opportunities for exploiting more interactions and improving performance for consolidated workloads, and iii) managing cache, bandwidth and prefetch jointly has the potential to outperform combinations of two of the techniques.

- A resource manger that dynamically manages cache partitioning, bandwidth partitioning and prefetch throttling in coordination considering the interactions between the three techniques. The solution works by employing individual resource managers to determine appropriate settings for each resource and a coordination mechanism to enable inter-resource trade-offs.

1.4 Organization of the thesis

The rest of the thesis is organized as follows. In Chapter 2, a summary of each paper is presented. Finally, Chapter 3 concludes the thesis, and discusses some possible future research directions.

Chapter 2

Summary of the Papers

2.1 Summary of Paper I

A cache partitioning solution consists of two parts: an allocation policy which decides the size of the partitions for each application and an enforcement mechanism to enforce them. Prior work have proposed allocation policies [3, 7, 8, 24, 25], but the drawback is their reliance on a centralized algorithm to determine allocations which limits how frequently this can be performed. Prior work on enforcement mechanism has shown that fine-grained partitioning, i.e. support many and varying partitions, is beneficial [5, 6, 9, 26–29] but the main shortcoming is that they do not take locality into account. A few proposals [7,8] have tried to address this but their solutions either require costly broadcasts or rely heavily on software support. Furthermore, the allocation policies in these proposals use a centralized algorithm which affects the overhead. A high overhead presents two problems: Firstly, limiting the frequency of reconfiguration which is important in order to adapt to application phase changes, and secondly, introducing unpredictable jitter in application execution.

An ideal cache partitioning, in the context of multi-core systems, needs to have a number of different characteristics. Firstly, it needs to be locality-aware and place partitions in a way which minimizes the on-chip distance. Secondly, it needs to support fine-grained partition sizes and to adapt quickly to application-phase changes, in order to at each instance of time have the most suitable allocation. Finally, it needs to have low enough overhead for performing allocation decisions and cause minimal OS intervention in order to be scalable.

In Paper I, we present DELTA, a scalable cache partitioning solution consisting of a fully distributed allocation policy and a locality-aware enforcement policy. In contrast to prior work, DELTA works with standard LRU-replacement policy and is implementable in hardware. DELTA's allocation policy consists of two parts: one intra-bank and one inter-bank mechanism. The inter-bank algorithm works by asynchronous exchanges of challenges among cores. The challenges include the potential performance benefit of increased cache capacity, and helps applications with larger performance benefit to gain more cache capacity. The intra-bank algorithm redistributes the cache capacity within a cache bank, giving a larger allocation to applications with a larger potential

performance gain. DELTA's enforcement mechanism combines bank- and way-level partitioning to enable fine-grained and locality-aware partitions. In order to locate data in the LLC a per-core Cache Bank Table (CBT) is used. The CBT contains mappings between addresses and cache banks, and it is accessed in parallel with the L2 cache in order to find the right LLC bank.

The distributed inter-bank allocation algorithm in DELTA uses two metrics, *pain* and *gain* as part of the asynchronous negotiation. The pain estimates the potential performance decrease with lost cache capacity while gain estimates the potential performance increase with additional cache space. The challenge message contains the potential gain that a given application would experience with additional cache capacity. The challenged cache bank compares its own pain with the gain of the challenging bank. If the gain is greater, a portion of the cache capacity belonging to the challenged bank is remapped to the challenger. The inter-bank allocation algorithm redistributes the cache capacity from applications which have a lower performance decrease when giving up cache capacity to applications which have a higher performance gain from additional cache space.

In Paper I, the proposal is evaluated with detailed simulations on both a 16- and 64-core tiled CMP using the Sniper simulator [30]. The performance of DELTA is compared against a private cache implementation, a shared NUCA implementation and an idealized centralized solution. The ideal centralized solution is used in order to evaluate the quality of the allocations performed by DELTA and uses the best known cache partitioning algorithm, Lookahead [3]. It does not model the overhead of computing allocation decisions. In the evaluation, an analysis of the overhead of calculating the allocations for DELTA and the best centralized algorithms is shown for different core counts. The analysis shows that the centralized algorithms overhead in time per invocation makes them unusable for large core counts with frequent reconfigurations.

The evaluation on a 16-core CMP shows that DELTA improves performance of on average 9% compared to an unpartitioned S-NUCA and by 6%, on average, compared to private caches (i.e., equal partitioning). The performance of the allocation using DELTA's is 2% lower than the idealized centralized solution. On a 64-core CMP, DELTA improves performance by on average 16% over an unpartitioned S-NUCA and is within 1% of the idealized centralized solution.

The evaluation in Paper I demonstrates that the distributed partitioning solution performs close to an ideal centralized solution. The distributed algorithm has low computational overhead which permits it to be implemented in hardware and allows for frequent reconfigurations while the enforcement scheme enables locality-aware placement.

2.2 Summary of Paper II

Multi-resource management combining either cache and bandwidth partitioning [17–19], or cache partitioning and prefetching [20, 21], or prefetch and bandwidth partitioning [22, 23], has been shown to be beneficial in reducing average memory access time and increasing performance. However, no study so far has considered coordinated management of all three techniques. Coordinately managing all three techniques provides several advantages. Firstly, it enables improving performance in more applications and addressing a broader range of application characteristics. Secondly, new trade-offs and interactions are enabled which have significant impact on performance.

In Paper II, we show an in-depth performance characterization of the applications in the SPEC CPU2006 suite. The results show that 90% of the applications have performance sensitivity (over 10% change in IPC) to at least one of the techniques, and 70% are also sensitive to multiple techniques. We make several observations regarding the interactions and trade-offs between the different techniques: i) the allocation of cache and bandwidth affects the performance impact of prefetching, ii) larger bandwidth allocation can compensate for inaccurate prefetches, iii) for an application sensitive to both resources the same performance can be gained, by either increasing cache size or enabling prefetching, or by either increasing bandwidth or cache allocation. In Paper II we also show, using exhaustive search, that for above 400 random workloads of four applications, coordinately managing three resources is better than any combination of two resources.

In Paper II, we present CBP, a coordinated mechanism for adaptive management of cache partitioning, bandwidth partitioning and prefetch throttling. The design is guided by observations and results from the performance characterization. CBP consists of three local controllers, one for each technique, and a coordination mechanism, see Figure 2.1. With CBP, the three local controllers are dynamically tuned and guided by heuristics, to give a good allocation of the resources, considering application characteristics and possible trade-offs and interactions. Recalibrations are performed periodically. First, cache space is allocated since altogether avoiding a memory access has higher impact than reducing the latency. As a next step, bandwidth is allocated based on the queuing latency of the memory requests and taking into consideration the cache allocation. Finally, the prefetcher setting is determined based on the current allocation of cache and bandwidth. The cache resource controller estimates

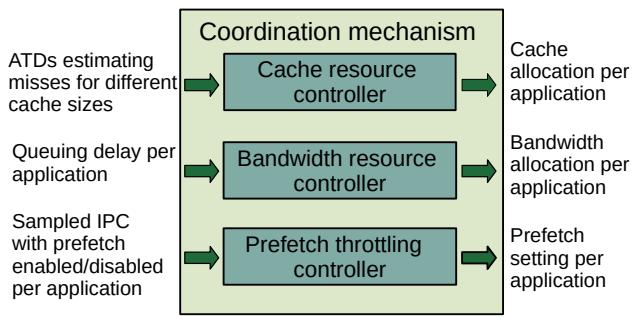


Figure 2.1: Overview of CBP resource manager.

the number of misses using auxiliary tag directories (ATDs) and allocates the capacity in a way that reduces the aggregate number of cache misses. The bandwidth resource controller uses the queuing delay per application in order to allocate the capacity, where the application experiencing the largest queuing delay will get the largest allocation. The prefetcher setting is determined by sampling, for a short time interval, the IPC with prefetching active/inactive and enabling the prefetcher only if the speedup is large enough.

In Paper II, CBP is evaluated against nine different single- and multi-resource managers. The baseline configuration represents an unpartitioned shared cache with unpartitioned bandwidth and prefetching disabled. The comparison points are CPpf [20] a state-of-the-art scheme combining cache partitioning and prefetching, equal partitioning of cache and bandwidth, as well as resource managers controlling only one resource or two of the three resources.

CBP is evaluated with multi-programmed workloads on a 16-core CMP using the Sniper simulator [30]. CBP improves performance by 11% on average compared to CPpf and by 50% on average compared to the baseline. We use the user-oriented fairness metric Average Normalized Turnaround Time (ANTT) in order to show that CBP does not increase performance at the cost of fairness. CBP increases fairness by 8% compared to CPpf and by 27% compared to baseline. According to the experimental results, the proposed multi-resource manager provides an effective solution for the main research problem. The evaluation shows that coordinately managing cache partitioning, bandwidth partitioning and prefetch throttling is better than any pair-wise technique and improves upon state-of-the-art.

Chapter 3

Concluding Remarks

Contention in shared resources, such as cache and bandwidth, can lead to destructive interference and large performance variations across workloads, detrimentally impacting average memory access time.

This licentiate thesis proposes adaptive resource management techniques to improve performance of multi-core architectures. Two different resource management techniques are presented. Paper I concerns resource management of the cache, where a distributed cache partitioning solution is proposed to avoid high computational overhead when determining allocations. The results show that it is possible to design a distributed solution for cache partitioning, which performs close to an idealized centralized solution. Paper II concerns multi-resource management and proposes a coordinated scheme with cache partitioning, bandwidth partitioning and prefetch throttling. Furthermore, the results for the coordinated multi-resource management scheme outperform any resource manager for two resources and improves upon state-of-the-art.

There are several interesting directions for future work. Firstly, one direction would be to target fairness or Quality of Service (QoS), instead of performance, as is currently done in both papers. Secondly, in the context of multi-resource management it would be interesting to study the scalability further. A possibility to further decrease the computational overhead would be to adapt the proposed single resource algorithms and make them distributed instead of relying on a centralized algorithm per resource. Thirdly, another future research direction would be to extend the multi-resource manager and also perform coordinated resource management for more resources such as core frequency, memory capacity and disk bandwidth, in order to increase performance further. The challenge with adding additional resources would be how to take the additional interactions and trade-offs into consideration without reaching a too high overhead for resource management. Finally, another interesting direction would be to explore and extend the resource management to the domain of multi-socket NUMA systems. Here, multiple multi-core architectures are connected by off-chip interconnects and memory is distributed across the systems. Being able to use underutilized cache/memory resources across sockets could give large performance impact. However, it is a challenging problem since bandwidth between sockets is limited and both on-chip and across-chip distances need to be taken into consideration.

Bibliography

[1] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger, "Clock rate versus ipc: The end of the road for conventional microarchitectures," *SIGARCH Comput. Archit. News*, vol. 28, no. 2, p. 248–259, May 2000.

[2] M. Bohr, "A 30 year retrospective on dennard's mosfet scaling paper," *IEEE Solid-State Circuits Society Newsletter*, vol. 12, no. 1, pp. 11–13, 2007.

[3] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *Proc. MICRO-49*, 2006.

[4] N. El-Sayed, A. Mukkara, P. A. Tsai, H. Kasture, X. Ma, and D. Sanchez, "Kpart: A hybrid cache partitioning-sharing technique for commodity multicores," vol. 2018-Febru, 2018, pp. 104–117.

[5] R. Manikantan, K. Rajan, and R. Govindarajan, "Probabilistic shared cache management (PriSM)," in *Proc. ISCA-39*, 2012.

[6] D. Sanchez and C. Kozyrakis, "Vantage: Scalable and Efficient Fine-Grain Cache Partitioning," *Proc. ISCA-38*, 2011.

[7] H. Lee, S. Cho, and B. R. Childers, "CloudCache: Expanding and shrinking private caches," in *Proc. HPCA-17*, 2011.

[8] N. Beckmann and D. Sanchez, "Jigsaw: Scalable software-defined caches," in *Proc. PACT-22*, 2013.

[9] X. Wang, S. Chen, J. Setter, and J. F. Martinez, "SWAP: Effective Fine-Grain Management of Shared Last-Level Caches with Minimum Hardware Support," in *Proc. HPCA-23*, 2017.

[10] J. Park, S. Park, M. Han, J. Hyun, and W. Baek, "Hypart: A hybrid technique for practical memory bandwidth partitioning on commodity servers." Institute of Electrical and Electronics Engineers Inc., 11 2018, pp. 1–14.

[11] D. R. Hower, H. W. Cain, and C. A. Waldspurger, "Pabst: Proportionally allocated bandwidth at the source and target," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017, pp. 505–516.

[12] F. Liu, X. Jiang, and Y. Solihin, "Understanding how off-chip memory bandwidth partitioning in chip multiprocessors affects system performance," in *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, 2010, pp. 1–12.

[13] B. Falsafi and T. F. Wenisch, "A primer on hardware prefetching," *Synthesis Lectures on Computer Architecture*, vol. 28, pp. 1–69, 5 2014.

[14] E. Ebrahimi, O. Mutlu, C. J. Lee, and Y. N. Patt, "Coordinated control of multiple prefetchers in multi-core systems," 2009, p. 316.

[15] F. Dahlgren, M. Dubois, and P. Stenstrom, "Fixed and adaptive sequential prefetching in shared memory multiprocessors," in *1993 International Conference on Parallel Processing - ICPP'93*, vol. 1, 1993, pp. 56–63.

[16] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt, "Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers," in *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, ser. HPCA '07. USA: IEEE Computer Society, 2007, p. 63–74.

[17] A. Sahu and S. Ramakrishna, "Creating heterogeneity at run time by dynamic cache and bandwidth partitioning schemes." Association for Computing Machinery, 2014, pp. 872–879.

[18] R. Bitirgen, E. Ipek, and J. F. Martínez, "Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach," 2008, pp. 318–329.

[19] J. Park, S. Park, and W. Baek, "Copart: Coordinated partitioning of last-level cache and memory bandwidth for fairness-aware workload consolidation on commodity servers," vol. 19. ACM.

[20] J. Xiao, A. D. Pimentel, and X. Liu, "Cppf: A prefetch aware llc partitioning approach." Association for Computing Machinery, 8 2019, pp. 1–10.

[21] G. Sun, J. Shen, and A. V. Veidenbaum, "Combining prefetch control and cache partitioning to improve multicore performance." Institute of Electrical and Electronics Engineers Inc., 5 2019, pp. 953–962.

[22] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt, "Prefetch-aware shared resource management for multi-core systems," *ACM SIGARCH Computer Architecture News*, vol. 39, p. 141, 2011.

[23] F. Liu and Y. Solihin, "Studying the impact of hardware prefetching and bandwidth partitioning in chip-multiprocessors." Association for Computing Machinery (ACM), 2011, p. 37.

[24] L. R. Hsu, S. K. Reinhardt, R. Iyer, and S. Makineni, "Communist, Utilitarian, and Capitalist Cache Policies on CMPs: Caches as a Shared Resource," in *PACT-15*, 2006.

[25] D. Thiebaut, H. S. Stone, and J. L. Wolf, "Improving disk cache hit-ratios through cache partitioning," *IEEE Trans. on Comp.*, 1992.

[26] H. Cook, M. Moreto, S. Bird, K. Dao, D. A. Patterson, and K. Asanovic, "A hardware evaluation of cache partitioning to improve utilization and energy-efficiency while preserving responsiveness," in *Proc. ISCA-40*, 2013.

[27] N. El-Sayed, A. Mukkara, P. A. Tsai, H. Kasture, X. Ma, and D. Sanchez, "KPart: A Hybrid Cache Partitioning-Sharing Technique for Commodity Multicores," in *Proc. HPCA-24*, 2018.

[28] G. Kurian, O. Khan, and S. Devadas, "The locality-aware adaptive cache coherence protocol," *ACM SIGARCH Comput. Archit. News*, 2013.

[29] Y. Xie and G. H. Loh, "Pipp: Promotion/insertion pseudo-partitioning of multi-core shared caches," in *Proc. ISCA-36*, 2009.

[30] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout, "An evaluation of high-level mechanistic core models," *ACM TACO*, 2014.

Paper I

DELTA: Distributed Locality-Aware Cache Partitioning for Tile-based Chip Multiprocessors

N. Holtryd, M. Manivannan, P. Stenström and M. Pericàs

Reprint from

IEEE International Parallel and Distributed Processing Symposium (IPDPS) 2020,
New Orleans, LA, USA, 2020.

2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)

DELTA: Distributed Locality-Aware Cache Partitioning for Tile-based Chip Multiprocessors

Nadja Holtryd, Madhavan Manivannan, Per Stenström, Miquel Pericàs
Department of Computer Science and Engineering
Chalmers University of Technology
Göteborg, Sweden
Email: {holtryd, madhavan, per.stenstrom, miquelp}@chalmers.se

***Abstract*—Cache partitioning in tile-based CMP architectures is a challenging problem because of i) the need to determine capacity allocations with low computational overhead and ii) the need to place allocations close to where they are used, in order to reduce access latency. Although, previous solutions have addressed the problem of reducing the computational overhead and incorporating locality-awareness, they suffer from the overheads of *centrally* determining allocations.**

In this paper, we propose DELTA, a novel *distributed* and locality-aware cache partitioning solution which works by exchanging asynchronous challenges among cores. The distributed nature of the algorithm coupled with the low computational complexity allows for frequent reconfigurations at negligible cost and for the scheme to be implemented directly in hardware. The allocation algorithm is supported by an enforcement mechanism which enables locality-aware placement of data. We evaluate DELTA on 16- and 64-core tiled CMPs with multi-programmed workloads. Our evaluation shows that DELTA improves performance by 9% and 16%, respectively, on average, compared to an unpartitioned shared last-level cache.

***Index Terms*—cache partitioning, multicore architectures, performance isolation**

I. Introduction

Efficient use of cache resources on chip multiprocessors (CMPs) is necessary in order to bridge the speed gap between processor and main memory. The last-level cache (LLC) is usually shared among all cores to maximize utilization. Unconstrained sharing, however, can result in destructive interference between workloads and lead to large performance variation, degrade throughput and violate per-application Quality of Service (QoS) requirements. Cache partitioning can mitigate destructive interference by isolating cache space between cores/applications.

A partitioning solution typically comprises two components: an *allocation policy*, to decide the size of the partitions for each application, and an *enforcement mechanism* to enforce the partitions. With regard to the allocation policy, known approaches target different objectives, e.g. to maximize throughput or improve fairness [1]–[3]. Utility-based cache partitioning (UCP) [1] aims to maximize throughput by assigning cache ways to applications that benefit most from the cache capacity. To do so, UCP leverages the Lookahead algorithm. The algorithm determines dynamic cache allocation based on marginal utility and partitions ways in a monolithic cache between applications but it has a high computational complexity. Approaches to determine cache allocations with lower computational overhead have been proposed [4]. However, as our evaluation of the overheads (described in Section IV) shows, the scalability of these proposals remains limited by their reliance on a centralized allocation algorithm. This reliance presents two problems. First, the execution time of the allocation algorithm limits the frequency of reconfiguration, especially as we scale to large core counts. And second, the invocation of the algorithm introduces unpredictable jitter in application execution. OS noise (jitter) has been identified as a major cause of both execution time unpredictability and untimely synchronization [5], [6]. This is particularly bad for multithreaded applications that rely on bulk synchronous parallelism (BSP) [6], [7]. As a consequence, centralized allocation approaches cannot be utilized when scaling to large core counts and requiring frequent reconfigurations.

Different enforcement mechanisms for cache partitioning have been proposed. Way and set partitioning are proposed in the context of monolithic caches with few cores [1], [2], [8], [9]. These schemes have the drawback of only supporting a limited number of coarse-grained partitions. Solutions have been proposed to enable fine-grained partitioning [10]–[17]. The main shortcoming of these techniques is that they do not take locality into account when partitioning LLCs in tiled CMPs. A few proposals have tried to address this limitation by enabling locality-aware placement [4], [18]. However, the allocation policies used in these proposals rely on a centralized allocation component and inherit its shortcomings.

An ideal cache partitioning solution should be fine-grained to support many and varying partitions, be locality-aware to place data close to where it is used, and adapt quickly to changes in application-phase behavior while still ensuring that allocation operations can be performed in a scalable manner, with low overhead and minimal OS intervention. We propose DELTA, a novel scalable cache partitioning solution for tile-based CMPs, that utilizes a *distributed* allocation policy and a locality-aware enforcement mechanism. In contrast to prior work, our solution uses a completely distributed and asynchronous allocation algorithm, works with a standard LRU-replacement policy, does locality-aware enforcement and is virtually transparent to the full software stack.

DELTA's Allocation Policy: DELTA's allocation algorithm comprises an inter-bank and an intra-bank component. The

1530-2075/20/$31.00 ©2020 IEEE
DOI 10.1109/IPDPS47924.2020.00066

inter-bank algorithm determines capacity allocations by asynchronously exchanging *challenges* among cores. A challenge represents the performance benefit of obtaining increased cache capacity and helps an application with larger performance potential to gain more cache capacity. The algorithm uses coarse-grained shadow-tags to collect reuse-distance information with minimal overhead, which is used as the basis for computing a challenge. The intra-bank algorithm periodically redistributes the cache capacity within a bank by giving more space to the application that has a larger potential performance gain. The distributed nature of the algorithm enables quick and incremental adaptation to program phase changes without requiring a central entity to determine and perform chip-wide reallocation of cache capacity for the different applications.

DELTA's Enforcement Mechanism: DELTA combines way partitioning and bank-level partitioning to achieve a locality-aware and fine-grained partition-enforcement mechanism. The partitioning solution uses per-core *Cache Bank Tables* (CBTs), where mappings between addresses and banks are recorded. When a request needs to access the LLC, the CBT is used to identify the cache bank that the address is mapped to. Inside each bank, way partitioning is used to divide the capacity. The flexibility of the enforcement mechanism makes it possible to keep data close to where it is used.

In summary, we make the following contributions:

(a) We propose DELTA, a fully distributed and locality-aware cache-partitioning solution. The distributed allocation algorithm asynchronously negotiates and makes effective cache allocation decisions. The flexibility of the enforcement mechanism enables locality-aware mappings. The distributed nature of the solution, coupled with low computational overhead, enables a hardware-based implementation. This allows the scheme to scale to large core counts while permitting frequent reconfigurations without invoking the operating system.

(b) We describe a novel allocation policy consisting of a coarse-grained and a fine-grained component which are responsible for carrying out inter- and intra-bank allocations, respectively. The inter-bank algorithm uses challenges to expand into multiple banks, while the intra-bank algorithm allows the allocation to grow within a bank.

(c) We present a reconfigurable NUCA enforcement mechanism that enables locality-aware mapping. The two-level mechanism combines coarse-grained, bank-level partitioning with fine-grained way partitioning. The CBT enforces flexible mapping of addresses to cache banks, which enables placing data close to where it is used.

We evaluate our solution on 16- and 64-core tiled CMPs. With multi-programmed workloads on a 16-core CMP we obtain speed-ups of up to 16% (geom. mean 9%) compared to an unpartitioned S-NUCA and up to 11% (geom. mean 6%) compared to private caches (equal partitioning). On the 64-core CMP DELTA improves performance by up to 28% (geom. mean 16%) over an unpartitioned S-NUCA.

The rest of the paper is organized as follows: Section II describes our proposed solution in detail. Section III discusses the methodology and Section IV presents the evaluation of the proposal. We provide an overview of related work in Section V and conclude in Section VI.

II. DELTA Cache Partitioning

Section II-A provides an overview of DELTA, followed by a detailed presentation of the algorithms and mechanisms in subsequent sections.

A. Overview

Both the allocation policy and the enforcement mechanism have an $inter\text{-}bank$ and an $intra\text{-}bank$ component. The inter-bank component takes care of the allocation and enforcement across cache banks, and the intra-bank part handles the allocation and enforcement within the banks.

DELTA allocation policy: Figure 1 provides an overview of the distributed allocation algorithm. The inter-bank allocation algorithm works by tiles periodically sending out challenge messages, as shown in Figure 1 (Step #1). The mechanism relies on two metrics called *pain* and *gain* (see Table I). A challenge message contains the potential *gain* that a given application would experience if it were to get additional cache capacity. The challenged tile compares its own *pain*, owing to predicted decrease in performance because of lost cache space, with the gain (Step #2), (see Section II-B2 for details about pain and gain). If the gain is greater, the challenged tile gives up space in the cache bank and informs the challenger tile with a response message (Step #3). A portion of the addresses ($[A_k - A_l]$ in Figure 1) belonging to the application running in the challenger tile is remapped to the cache bank in the challenged tile (Step #4). The addresses that have been remapped to a different bank are then invalidated in their previous location. The inter-bank allocation policy helps applications acquire additional cache capacity by mapping data to cache banks in other tiles.

The second part of the allocation algorithm, the intra-bank algorithm, governs changes within a bank. The intra-bank algorithm is invoked periodically in every cache bank. In each interval some ways are transferred to the partition with most gain from the partition with the least. This way, reassignment has little overhead since it does not affect the mapping of addresses to banks, and therefore does not lead to invalidations. While the inter-bank allocation algorithm helps an application to expand its working set into other tiles and get a fixed capacity, the intra-bank algorithm helps in fine-tuning the capacity in banks that an application has already expanded

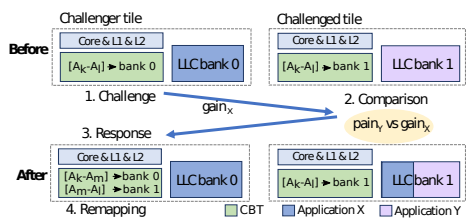


Fig. 1: Overview of steps in DELTA allocation.

$Gain$	Predicted performance increase due to increased cache space
$Pain$	Predicted performance decrease due to lost cache space

TABLE I: Terminology

into. The intra-bank algorithm reports information about ways that an application wins/loses in a cache bank outside the tile and this acts as a feedback to guide inter-bank expansion. The details about the DELTA allocation algorithm are discussed in Section II-B.

DELTA enforcement mechanism: The inter-bank mechanism uses a mapping table as shown in Figure 1, the CBT, to map addresses to cache banks (see Section II-C for details). On a private cache miss the CBT provides the mapping of each address to the LLC bank where it resides. The mappings in the CBT are changed when reconfigurations are triggered by the allocation algorithm. The flexible mapping enabled by the CBT is in contrast to S-NUCA which maps addresses to cache banks statically. The intra-bank mechanism relies on hardware support for way partitioning (discussed in Section II-C) similar to that available in some commodity systems [19].

B. DELTA Allocation

1) Allocation Algorithm: Inter-bank allocation: The pseudo-code for the inter-bank allocation algorithm is shown in Algorithm 1. Each tile (Challenger) starts by computing the pain and gain at the beginning of every inter-bank reconfiguration interval (i_{inter}), which is set to 1ms. An analysis motivating this interval is presented in Section IV-D. A challenge message is only sent if the calculated gain is above a threshold and the size of the allocation is larger than the minimum limit (line 4). The requirement to be above the minimum allocation limit is to avoid placing data far away instead of expanding in the home bank. The choice of which tile to challenge (Challenged) is based on the distance to the tile. Each tile will start by challenging the closest neighbouring tiles, with a hop distance of one, before choosing tiles further away (line 5). A single challenge is issued by every tile in each (i_{inter}) interval if it satisfies the preconditions. The algorithm will only pick a particular tile for a second challenge after it has exhausted other candidates, regardless of whether the previous attempt was successful.

When a challenge is received, the gain from the challenger tile is compared to the pain of the challenged tile. The algorithm uses pain, instead of gain, for comparison in order to accommodate the potentially high impact on performance for the application running on the challenged tile. Furthermore, it also acts as a deterrent to prevent one tile from easily invading and taking over the capacity of neighbouring tiles. In case an application running on tile A is sharing its cache bank with another running on tile B and receives a challenge from a different application running on tile C, the algorithm will compare the Pain_A, Gain_B and Gain_C to determine if the challenge is successful (line 10). In case the challenge is successful, a fixed capacity (number of ways) is allocated in the challenged tile and a response is sent to the challenger tile as a notification (line 12-13). On receiving a successful response the CBT is updated and invalidations are triggered if required (line 18-20). The intra-bank algorithm, described later in this section, discusses how the allocation for the challenger tile can grow to encompass the entire cache bank gradually over time. If the challenged tile does not use its home cache bank (i.e. the core is idle), the algorithm will allocate the whole cache bank to the challenger tile immediately instead of gradually. This is done to make it easier for applications that are running alone to increase their allocation quickly as the cache banks will otherwise remain underutilized.

Algorithm 1: Inter-bank allocation pseudo-code

Input: mlp, allocationForChallanger, interDeltaWays

```
1  In tile Challanger at time period i_inter ;
2  pain = calculatePain(mlp, allocationForChallanger);
3  rawGain = calculateRawGain(mlp,
     allocationForChallanger);
4  if rawGain >gainThreshold AND allocationForChallanger
     >minWays then
5  |  challenged = getClosestNeighbour();
6  |  gain = rawGain / distanceTo(challenged);
7  |  challenge(challenged, challenger, gain);
8  end
9  In tile Challenged on receiving a challenge;
10 partition = partitionWithSmallestGainOrPainInChallenged(
     challengedPain, challengerGain,
     gainChallengedPartitions);
11 if partition then
12 |  updateWayPartition(challenger, partition,
   |    interDeltaWays);
13 |  respondWithNewPartition(challenged, challenger,
   |    true);
14 else
15 |  respondWithNewPartition(challenged, challenger,
   |    false);
16 end
17 On response;
18 if success then
19 |  updateCacheBankTableWithNewPartitionIn(challenged);
20 |  invalidateAddressesWithChangedBankPlacement();
21 end
22 markAsChallenged(challenged);
```

Intra-bank allocation: The pseudo-code for the intra-bank algorithm is shown in Algorithm 2. This is triggered in each tile at every intra-bank reconfiguration interval (i_{intra}) which is set to 0.1ms. The algorithm works by comparing the gain for each partition that shares the cache bank (line 2-3) and reassigns some ways ($intraDeltaWays$) from the partition that has the least gain to the one that has the most (line 5). Here, unlike the inter-bank allocation, the comparison only considers the gain of every application to determine the winner, for two reasons. Firstly, the application running on the tile must have already demonstrated a significant gain, more than the home bank's pain, to have been allowed to expand into a different tile. Secondly, intra-bank changes in allocation are lightweight and do not introduce any invalidation-related overheads (except when leaving a tile). In case an application running on tile A is sharing its own cache bank with two others running on tile B and C, a comparison will happen between Gain_A, Gain_B and Gain_C, to determine which contending

Algorithm 2: Intra-bank allocation pseudo-code

```
   Input: intraDeltaWays
1  In each tile at time period i_intra;
2  partitionSmallest =
     partitionWithSmallestGain(gainsOfPartitionsInBank,
     minWays);
3  partitionLargest =
     partitionWithLargestGain(gainsOfPartitionsInBank);
4  if partitionWithLargestGain !=
     partitionSmallestGain then
5      updateWayPartitioning(partitionLargest,
         partitionSmallest, intraDeltaWays);
6      reportNewAllocation(partitionSmallest,
         partitionLargest);
7  end
```

application will win/lose cache ways.

After reassignment the information about the number of ways are sent back to the respective contending home tiles (line 6). This acts as a feedback mechanism between the intra- and inter-bank algorithm since the current allocation is an important factor in determining the pain and gain, as will be described in the next section. The inter- and intra-bank algorithms are invoked periodically after an initial state where cache capacity is equally partitioned among all tiles.

2) Computing Gain and Pain: The measure of pain and gain is based on simple heuristics that lead to effective allocation decisions. We use gain to predict how an application will react to an increase in cache capacity ($gainWays$) by expanding in/to other tiles. In order to compute gain we take into consideration information about the number of misses provided by the shadow tags, memory-level parallelism (MLP), current capacity allocation outside the home tile and the hop distance. The potential gain for an application running on tile i and expanding into tile j is calculated using the following formula:

$$Gain_{\text{i,j,gainWays}} = \frac{a_{gainWays} * (k+1)^{-1}}{m * (l+1)} \quad (1)$$

where a is the number of misses that potentially can be avoided with $gainWays$ additional ways, k is the number of ways outside of the home tile, m is the MLP of the application running on tile i and l is the hop distance from tile i to j.

The rationale behind factoring in the aforementioned attributes in the gain expression is as follows. Firstly, factoring in the number of avoidable misses provides an estimate of reduction in the number of long-latency memory accesses which influences performance. This value can be read directly from the shadow tags in the monitoring hardware for a given core. MLP is factored in because this coupled with the number of misses helps to get a better estimate of the performance impact of cache allocation decisions. The MLP estimate is obtained through performance counters. Lastly, we factor in the current allocation in remote tiles and the hop distance to introduce fairness and ensure that no single application expands its allocation too aggressively.

We use pain as a heuristic measure to predict how an application will react to losing available cache capacity ($painWays$) on the home tile where it is running. The pain value is never communicated to other cache banks. In order to compute pain we only take information about misses provided by the shadow tags and MLP into account. Unlike the formula for gain, we do not take information about allocations outside home bank and the distance into account because the goal here is to protect the capacity allocation in the home tile where the application is running. Since the pain is not scaled it will grow faster, if there are more misses, which will enable the home bank application to protect its allocation. The pain of losing $painWays$ for the application running on tile j is calculated using the following formula:

$$Pain_{\text{j,painWays}} = \frac{a_{painWays}}{m} \quad (2)$$

where a is the number of misses that will be incurred if the allocation is decreased with $painWays$ and m is the MLP. Our evaluation in Section IV shows that the pain/gain heuristics leads to good cache allocation decisions. We leave further optimization of the pain and gain measures used in this study for future work.

3) Monitoring hardware: We adopt Qureshi's UMON sampled tag array [1] in this work. The original UMON mechanism can predict the number of cache misses under all possible cache allocations (at a single way granularity) based on the access pattern the application has exhibited before. The coarse-grained UMONs, that we use, work by tracking the number of accesses to a shadow tag at coarse-granularity (corresponding to 4 ways). The number of tags required will still be the same but the associated way-hit counter overheads are reduced. The solution also uses dynamic set sampling to decrease the overhead of the monitoring hardware, like the original proposal.

4) Hardware-based implementation: The inter-bank and intra-bank algorithms are implemented in hardware owing to its low computational complexity (see Section IV-E for details). To implement the algorithms each LLC bank controller is provisioned with an ALU capable of computing the pain and the gain and for comparing the values. The inter-bank scheme requires, for each bank, a register array with $N+2$ entries, with $log_2(N)$ bits per entry, to store the pain values of other banks. In addition, each bank also includes a register array with $N+1$ entries, with $log_2(N)$ bits per entry, to store the id of other tiles in increasing order of distance to determine the next tile to send the challenges to. The intra-bank algorithm leverages the state used by the inter-bank algorithm for establishing allocations.

C. DELTA Enforcement

DELTA's enforcement mechanism has two components. The inter-bank enforcement mechanism utilizes a CBT (the detailed design is presented later in this section) which contains the mapping between address ranges and cache banks. The CBT permits the allocations to span multiple banks by mapping portions of the address space to different banks. The CBT is accessed in parallel with the L2 cache to determine in which LLC bank a certain address is mapped to. The

intra-bank enforcement mechanism comprises a standard way-partitioning (WP) unit that keeps track of which ways in the cache bank each core is allowed to insert lines into.

1) Bank partitioning: Each core has a CBT which is organized as a small fully-associative range table that holds the information of address range to bank translation. The CBT works with physical block addresses. We use a simplified version of the range based organization proposed by Gandhi et al. [20]. The total amount of storage required for each CBT is $log_2(N) \times N$ bits, where N is the maximum number of distinct ranges, which is equal to the number of cores/banks. The number of used entries (i.e. ranges) in the CBT is equal to the number of LLC banks allocated by the local core. Only in the rare scenario in which a a single core is active, can a core's CBT grow to map the majority of the chip resources. Therefore, in practice this value is much smaller than the total number of banks and the cost of the associative look-up is negligible.

The CBT is updated when the allocated capacity for a tile expands to/retreats from another tile. When the CBT is updated there is a remapping of address ranges to cache banks. The size of the address range mapped to a bank is proportional to the size of the allocation in that cache bank. Examples illustrating when and how the CBT is updated as allocations expand/retreat are presented in Section II-D.

There are two important design choices for the CBT: (i) how many bits to use, and (ii) which bits to use, i.e. how to map addresses to cache banks. We evaluated different options and found that by using just 8 bits following the set index, as shown in Figure 2, it is possible to effectively distribute the footprint of the application across the different banks. We reverse the bits before we index into the CBT to obtain the bank mapping. The reversing operation turns the least significant bits with the highest entropy to the most significant, which proves to be a reasonable solution for mapping addresses uniformly for the applications we consider.

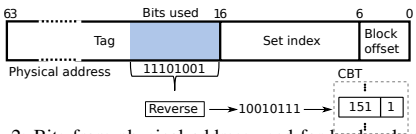


Fig. 2: Bits from physical address used for bank selection.

2) Way partitioning: During insertion, the WP unit uses a bitmask to indicate which cores can insert into a given way in a cache bank. All cores can however access data irrespective of which way it resides. Way-partitioning enforced using bitmasks is practical and has been implemented in commodity systems [19]. The total amount of storage required for each WP unit is $N \times W$ bits, where N is the number of cores and W is the number of LLC ways. DELTA can also work with other fine-grained intra-bank partitioning schemes proposed in literature [14], [15], [21].

3) Invalidation support: A common strategy to handle change in the mapping of an address to a LLC location is to invalidate the line in the cache bank where it currently resides. This invalidation is done by flushing the cache line. Several commodity systems provide ISA level support for this [22]. This is widely used by page coloring mechanisms [23]. However when remapping a large range there is increased overhead due to additional instructions needed for invalidating each address. We therefore rely on hardware support for performing bulk invalidation efficiently. The bulk invalidation unit works by checking the tags to identify addresses that fall in the specified range and invalidates them. This approach does not incur the instruction overhead of cache flushes.

D. Putting it all together

We clarify how the partitioning solution works with the help of two examples that illustrate the different use cases.

Example 1, Inter-bank expansion. The capacity allocation expands to a different tile when a challenge is successful. In Figure 3 we show the process of expansion into a new tile, as well as the state of the CBT and WP before/after the change. We assume that tile 4 has capacity allocated in cache banks in tile 4 and 0, as indicated in the CBT for tile 4. Since, the core in tile 4 sees a considerable gain from expanding its allocation it issues a challenge to tile 5 (#1). Tile 5 compares the gain coming from tile 4 with its own pain of losing cache capacity (#2). Since the gain for tile 4 is considerably larger, tile 5 decides to assign $interDeltaWays$ of ways from its allocation to tile 4 and updates its WP unit (#3). In this case, ways 12-15 are assigned to tile 4 and a response message is sent to tile 4. On receiving the message the tile updates its CBT to also include tile 5 (#4). This is followed by remapping addresses in range 192-255 from tile 4 to tile 5, and invalidating them where they were previously located (#5). Note that expansion process does not require invalidations in tile 5.

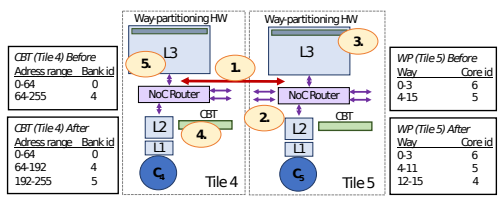


Fig. 3: Example of expansion.

Example 2, Intra-bank algorithm and retreat. The intra-bank algorithm determines whether allocations within a bank expand or shrink. The decision on which partition expands or shrinks is based on the gain of the different applications that share the cache bank. Whenever a partition expands or shrinks the WP unit is updated to reflect the new allocation for the partitions. A shrink will result in a retreat if a partition loses all the ways it was assigned in the cache bank. This scenario is shown in Figure 4. After the intra-bank algorithm is triggered in tile 5 the algorithm decides that tile 4 must give up the entire capacity in the cache bank since it has the least gain among

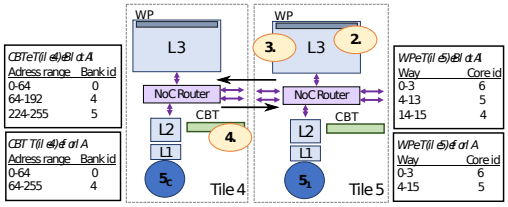


Fig. 4: Example of retreat.

those sharing the cache bank (#1). As a consequence, the WP unit is changed to show the new configuration where ways 14-15 are reassigned to tile 5. The change triggers a message back to tile 4 informing about the retreat. The CBT in tile 4 will now be updated (#2). The addresses previously mapped to tile 5 are remapped, in this example addresses in range 224 to 255 are now remapped back to tile 4. All addresses in the range will consequently be invalidated in tile 5 (#3).

The detailed evaluation of DELTA is presented in Section IV. We show in Section IV-E that the overheads introduced by DELTA are marginal.

E. Support for multithreading

When running multi-threaded workloads containing shared data, accesses to the same line from two different tiles will end up inserting the blocks in two different LLC banks, breaking coherence. To address this, we propose to distinguish between private and shared data at a page granularity and handle them differently. Detection of shared pages including cross-process sharing is performed by the TLB. We adopt a classification scheme, proposed in R-NUCA [24] and used in other NUCA schemes [4], to dynamically classify pages as private or as shared incrementally and lazily. Lines belonging to shared pages are mapped to the cache banks using a fixed S-NUCA strategy whereas lines belonging to private pages are mapped to banks based on the mappings available in the CBT. When a page is first classified as shared all the lines belonging to the page are invalidated.

The allocation algorithm also needs a minor change. On receiving a challenge, the processIDs of the different threads are compared, and the challenge will only be successful if they are different. The rationale is to not let threads from the same application (homogeneous multi-threaded) compete for capacity since it can adversely impact application progress. We expect the performance of this extension with multi-threaded application to be similar to R-NUCA since private data and most of shared data are dealt with in a similar way. Multi-threaded workloads are analyzed in Section IV-C.

III. Experimental Methodology

A. Simulated Architecture

We evaluate our proposal on a 16/64 core tiled CMP architecture modeled using the Sniper Simulator [25]. Details about the baseline architecture are shown in Table II. Each tile has an out-of-order (OOO) core with a private L1 data and instruction cache, a unified private L2 cache and a LLC bank of 512KB. The cache latencies assumed have been modelled using CACTI 6.5 [26].

Cores	16 / 64 cores, x86-64 ISA, 4GHz, OOO, Nehalem-like, 128 ROB entries, dispatch width 4
L1 caches	32KB, 8-way set-associative, split D/I, 1-cycle latency
L2 caches	128KB private per-core, 8-way set-associative, inclusive, 6-cycle data and 2-cycle tag latency
LLC	512KB per-tile, 16-way set-associative, inclusive, 9-cycle data and 2-cycle tag latency, LRU
Coherence protocol	MESIF-protocol, 64 B lines, in-cache directory
Global NoC	4x4 / 8x8 mesh, 4-cycles hop latency (3-cycle pipelined routers, 1-cycle links)
Memory controllers	4 / 8 MCUs, 1 channel/MCU, latency 80 ns, 12.6GB/s per channel
DELTA parameters	reconfiguration interval i_{inter}=1ms i_{intra}=0.1ms, $gainThreshold$=0.5, $minWays$=4, $interDeltaWays$=4, $intraDeltaWays$=1, $gainWays$=4, $painWays$=4

TABLE II: Configuration of the simulated 16- and 64-core tiled CMP.

In Section IV-E we demonstrate that state-of-the-art allocation algorithms, Lookahead or Peekahead, cannot compute locality-aware allocations in a scalable manner. This is because the time needed to compute allocations and locality-aware placement far exceeds the 1 ms reconfiguration interval that we target in this study especially as we scale to larger core counts. In order to fairly compare our distributed solution against the centralized solutions, we model an *ideal centralized* solution that calculates both allocations and locality-aware placement in zero time (no overhead). The ideal solution represents an upper bound on dynamic allocation decisions using the best known centralized algorithm, Lookahead. We use Lookahead as a reference since Peekahead too computes the same allocations as Lookahead albeit with lower overhead. The ideal centralized scheme uses the DELTA enforcement mechanism to support locality-aware mapping in banked LLCs. UMONs are used in each core to measure misses for all possible cache capacity allocations for each application. The cost of invalidations that occur due to remapping of addresses to banks (invalidation+re-fetch) are modelled in detail for both DELTA and the ideal centralized scheme. In addition, we also evaluate an unpartitioned, static NUCA implementation with line-interleaved LLC addresses (unpartitioned S-NUCA), and private LLC, with equal static partitioning of capacity per core (private) for comparison.

DELTA dynamically considers allocations in increments of 32KB from a cache size of 128KB up to 6MB (per application) for the 16-core configuration and 128KB to 24MB for the 64-core case. Each core reserves a minimum of 128KB (see Table II) in the LLC to avoid potential back-invalidations due to the inclusive cache hierarchy.

B. Workloads

We use the entire SPEC CPU2006 suite in our evaluation. The applications are in the format of whole program pinballs [27]. Workload mixes are constructed by classifying applications in one of the four categories - *cache-insensitive*,

cache-sensitive low, *cache-sensitive low medium* and *thrashing*, depending on sensitivity to different cache sizes. The classification is performed by running each application for 1B instructions (after fast-forwarding for 1B instructions) with cache sizes of 128KB, 512KB and 8MB. The applications that show improvement in IPC of over 10%, as the cache size increases, are classified as sensitive for a particular cache-size region. Sensitive applications that show improvement in the 128KB to 512KB region are classified as cache-sensitive low and those that also show improvement in the 512KB to 8MB are classified as cache-sensitive low medium. The detailed classification is presented in Table III. Lastly, cache insensitive and thrashing applications experience less than 10% improvement in the 128KB to 8MB range. Among these, we classify applications with a number of Misses-Per-Kilo-Instruction (MPKI) above five as thrashing and the rest as cache insensitive.

Insensitive (I)	povray(po),sjeng(sj),namd(na), zeusmp(ze),GemsFDTD(Ge)
Thrashing (T)	bwaves(bw),libquantum(li),milc(mi)
Cache-sensitive low (L)	h264ref(h2),gromacs(gr),astar(as), gamess(ga),lbm(lb),tonto(to), wrf(wr),leslie3d(le),hmmer(hm)
Cache-sensitive low medium (LM)	dealII(de),omnetpp(om),xalancbmk(xa), gobmk(go),bzip2(bz),gcc(gc),mcf(mc), soplex(so),perlbench(pe),sphinx3(sp), calculix(ca),cactusADM(cac)

TABLE III: Classification of SPEC CPU2006 benchmarks.

Name	Composition	Benchmarks
w1	LM	de,om(2),pe,ca,bz,go(2),ca,hm,le,go,bz,gc,so,mc
w2	L+LM	bw,sj,na,ze,li,mi,ca,sp,de,om,go,go,bz,gc,mc,pe
w3	T+L	to(2),bw(3),lb(2),li(3),h2,mi,gr,as,ga,mi
w4	T+LM	delII,bw(3),so,li(2),hm,pe,mi(3),go,om,bz,go
w5	I+L+LM	gc,po,Ge,as,pe,wr,ga,cac,to,hm,sj,h2,bz,ze,gr,so
w6	I+T+L+LM	na,de,li,gr,wr,so,mi,as,mi,to,ze,om,bw,h2,Ge,hm
w7	I+T+LM	sj,bw(2),bz,wr,li(2),gc,mi,de,na,om,ze,mi,go,Ge
w8	I+T+L	po,bw(2),h2,sj,li(2),gr,na,mi,as,Ge,ga,wr,lb,mi
w9	I+LM	po,om,sj(2),go,na(2),le,ze,go,Ge,bz,wr,ca,sp,gc
w10	I+L	po,to,sj,h2(2),na,lb(2),ze(2),gr,Ge,as,wr,ga,po
w11	T+L+LM	sp,bw,h2,om,li,gr,go,mi(2),as,hm,bw,ga,le,lb,calulix
w12	random	go,lb,ca,sp,bw,go,li(2),ga,h2,ze,to,so,gr,mi,pe
w13	random	lb,to,pe,go,gc,mi,li(2),na,h2,cac,ze(2),ca,so,as
w14	random	de,bw,mc,li,pe,mi,ca,wr,go,po,hm,na,go,ze,so,Ge
w15	random	to(2),po,lb,li,mi,lb,wr,h2,sj,gr,na,as,ze,ga,Ge

TABLE IV: Workload mixes.

We construct a total of 15 workload mixes by combining the applications from the categories described above. Applications from each category are picked randomly while not allowing duplicates unless all applications in a category have already been picked. Details about workload mixes are presented in Table IV. We construct workload mixes for 64 cores by replicating the 16-core workload four times. The applications in a workload mix are mapped to cores randomly.

C. Methodology

We fast-forward for 8B/2B instructions for the 16/64 core simulations. Detailed simulations are carried out until all benchmarks have completed at least 500M/125M instructions and statistics are reported based on the first 500M/125M instructions for each application. We simulate fewer instruction in fast-forward and detailed mode for 64-core CMP to reduce simulation time. The methodology is in line with earlier works [4], [14], [15].

D. Metrics

We use IPC as a measure of performance. We report the geometric mean of IPCs of the applications in a workload, as a performance metric for the workload. We also report the following fairness and throughput metrics: average normalized turnaround time (ANTT) and system throughput (STP) [28]. ANTT is given by $\frac{1}{N}\sum_{i=1}^{N}\frac{CPI_i}{CPI_{i,private}}$ and STP is given by $\sum_{i=1}^{N}\frac{CPI_{i,private}}{CPI_i}$ ANTT and STP are commonly used for performance evaluation of multi-programmed workloads.

IV. Evaluation

We first compare DELTA against alternative cache organizations and allocation algorithms (described in Section III-A). Next, we show results for multithreaded applications followed by the impact of reconfiguration frequency. Finally, we provide an analysis of DELTA's overheads.

A. Multi-programmed mixes on 16-core CMP

Figure 5 shows the performance for multi-programmed mixes normalized to the unpartitioned S-NUCA. On average, DELTA improves performance by 9% (up to 16%) over S-NUCA whereas the ideal centralized solution shows an average improvement of 12% (up to 22%). In comparison, the private scheme shows an average improvement of 3% over S-NUCA. The results for comparing the fairness and throughput of DELTA and the ideal centralized scheme are shown in Figure 6. On average, DELTA is 2% behind in terms of ANTT and 5% behind in terms of STP, than the ideal centralized scheme. Note that a lower value signifies greater fairness with ANTT while a higher value is equivalent to larger throughput with STP.

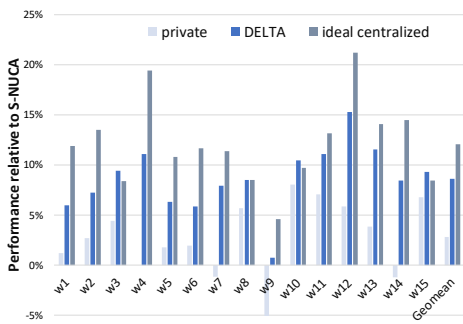


Fig. 5: Performance of workload mixes normalized to unpartitioned S-NUCA on a 16-core CMP.

As can be seen in Figure 5, the ideal centralized scheme is better than DELTA in 11 out of 15 mixes. In four cases DELTA performs on par or better. In order to understand the performance gap between the ideal centralized scheme and DELTA we investigate a single workload in detail. Figure 7 shows the performance of different applications in a single

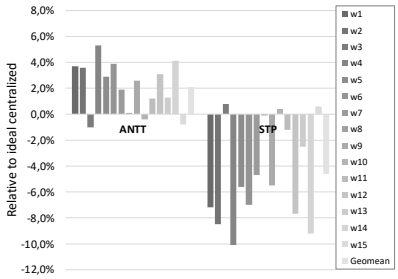


Fig. 6: Fairness comparison between ideal centralized and DELTA.

workload using the ideal centralized scheme, normalized to DELTA (in gray). In addition, it also shows the performance of the private scheme normalized to DELTA (in blue). As can be seen in the figure, most of the applications perform almost identically with the exception of xalancbmk and soplex. For these two applications, the ideal centralized solution performs considerably better than DELTA, by 45% and 35%, respectively. Note that DELTA performs better than the private scheme for these two applications by 12% and 36%.

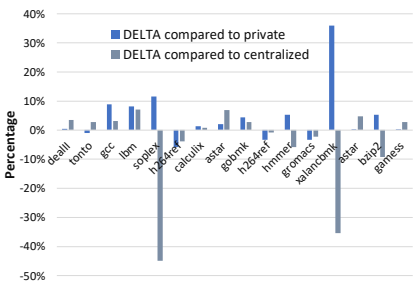


Fig. 7: Normalized performance for applications in *w2* on a 16-core CMP.

To understand the trend for xalancbmk and soplex, we compare the allocations of cache capacity, in terms of number of ways, made by the ideal centralized scheme and DELTA, for the different applications in the workload. The ideal centralized algorithm gives a larger allocation of 42 respectively 50 ways on average to xalancbmk and soplex in comparison to DELTA which gives 26 and 20 ways. This behaviour can be attributed to the *farsighted* nature of the ideal centralized scheme i.e. it uses information about the entire miss curves for all applications to determine allocations. DELTA, in contrast, is *nearsighted*, i.e. uses a limited window of the miss-rate curves to determine the pain/gain which influences allocations. As xalancbmk and soplex do not see a considerable improvement in the limited window, DELTA does not allocate as much cache capacity as the ideal centralized scheme. The difference in the size of allocations impacts the performance because these applications are sensitive to additional cache capacity.

In Figure 8 we show the performance of individual applications in one of the workloads where DELTA is on par with the ideal centralized scheme. We see that individual applications mostly perform as well as or better than the centralized scheme even though DELTA is nearsighted. The same trend holds also for the other workloads (w3,w8,w10,w15).

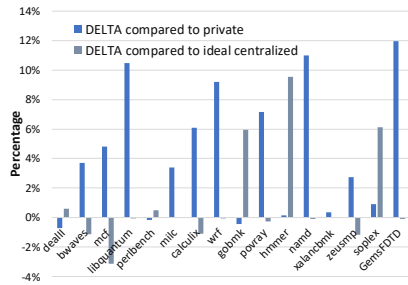


Fig. 8: Normalized performance for applications in *w3* on a 16-core CMP.

B. Multi-programmed mixes on 64-core CMP

To investigate how our proposal scales to larger core counts we evaluate a 64-core CMP. Figure 9 shows the performance for the individual multi-programmed workload mixes. DELTA, on average, improves performance by 16% (up to 28%) over S-NUCA, while the ideal centralized scheme improves performance by 17% (up to 35%). The private scheme performs better for 64-cores than for 16-cores, but is generally regarded as an inefficient solution since it cannot handle underutilized scenarios. The results for comparing fairness and throughput between ideal centralized and DELTA indicate that the difference between the two schemes is 1% for STP and less than 1% for ANTT (not shown). The results also indicate that DELTA makes good allocation decisions, on par with an ideal scheme, in spite of the distributed nature of the algorithm that increases the number of re-configurations (steps) required to span across all the banks in a CMP.

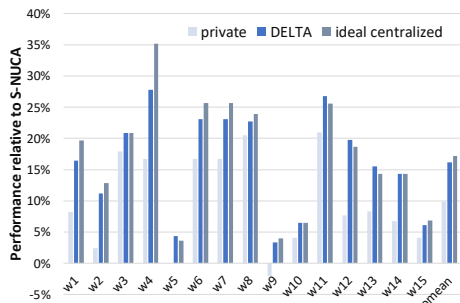


Fig. 9: Performance of workload mixes normalized to unpartitioned S-NUCA on a 64-core CMP.

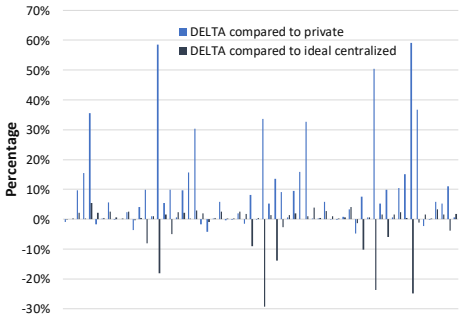


Fig. 10: Normalized performance for applications in *w2* on a 64-core CMP.

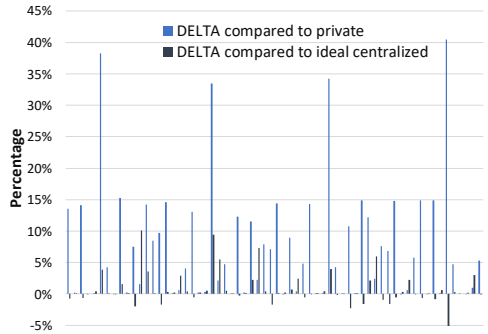


Fig. 11: Normalized performance for applications in *w13* on a 64-core CMP.

The performance gap between DELTA and the ideal centralized scheme has diminished on average, compared to the results for a 16-core CMP. For seven workloads (w3, w5, w10, w11, w12, w13, w14) DELTA is on par or better than ideal centralized. For workload w2, as shown in Figure 10, we observe that DELTA falls behind the ideal centralized scheme in the 64-core CMP experiments, similar to the trend seen with the 16-core CMP. For many applications in this workload DELTA is still better than the ideal centralized scheme but in the few cases where the reverse is true, the difference in performance is comparatively larger. For applications such as xalancbmk and soplex the ideal centralized scheme surpasses DELTA by giving a larger allocation, because it is farsighted.

We investigate one of the workloads (w13) where DELTA performs better than the ideal centralized scheme, as shown in Figure 11. The farsightedness of the ideal centralized scheme results in it allocating over 250 ways to applications such as lbm and libquantum. Moreover, the centralized scheme does not consistently detect the benefit of giving these applications a large allocation, and switches the cache capacity allocation between a large and small allocation. In general, giving larger allocation to a few applications puts severe constrains on the allocations for the other applications and degrades the overall performance. DELTA does not suffer from making these unadvantageous allocations and performs better for the mixes containing applications like lbm and libquantum.

In summary, the evaluation shows that DELTA performs almost as well as the ideal centralized scheme as we scale to 64 cores. Furthermore, this demonstrates that a dynamic distributed scheme can give good allocations, without incurring the overheads associated with computing allocations centrally.

C. Multi-threaded applications

We estimate the performance of DELTA using SPLASH2 suite in order to understand how the scheme performs with multithreaded applications. Figure 12 shows the speed-up obtained by DELTA over the S-NUCA implementation and compares it to the private cache configuration. We execute each application on the 16-core CMP and using large input sets (from Sniper) to obtain performance data for the baselines. We use number of cycles for the longest running thread within the parallel region, which we identify as the region of interest (ROI), as a measure of performance.

We follow a two step process to estimate the performance of DELTA. Firstly, we measure the ratio of private/shared pages and cache blocks. These results are shown in Table V. For this we develop a pintool [29] that instruments all loads and stores in the region of interest to measure inter-thread sharing at page and cache block granularity. Next, we estimate the performance of DELTA by performing a piece-wise reconstruction of the execution in which private accesses are modeled according to private LLC baseline's performance, and shared accesses are modeled according to the S-NUCA baseline performance. To simplify, we assume that the LLC accesses are uniformly distributed across pages. Private pages are reclassified at most once, and the S-NUCA mapping is never reverted. Hence, for long running applications this overhead is negligible. We expect the estimation to be accurate since DELTA maps lines from private-pages to the private bank and utilizes S-NUCA mapping for lines from the shared pages.

By design (see Section II-E), the performance of DELTA is usually between the performance of the S-NUCA and private baselines, depending on the amount of private/shared pages. Over the entire SPLASH2 suite, the average performance of DELTA compared to both the private LLC configuration and S-NUCA configuration is within 1% for both cases. The actual

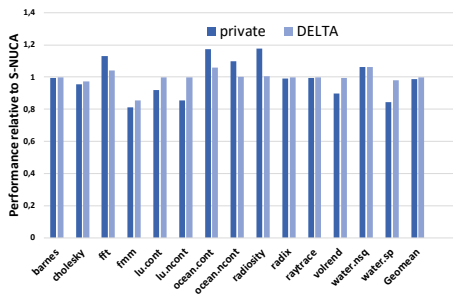


Fig. 12: Normalized performance for splash2 on a 16-core CMP.

App.	barnes	cholesky	fft	fmm	lu.cont	lu.ncont	ocean.cont
Page	8.2	62	33	73	0.5	0.5	38
Block	9.3	66	34	65	0.3	0.3	98.6
App.	water.sp	radiosity	radix	raytrace	volrend	water.nsq	ocean.ncont
Page	10	3	5.2	17	5.7	99.8	1.1
Block	70	4.6	10	24	21	91	99

TABLE V: Percentage of private pages and blocks.

performance for each benchmark depends considerably on the amount of sharing, with variations of up to 20%. For example, in `lu.ncont`, which has high ratio of sharing (>99%), DELTA's performance is almost equal to the S-NUCA performance, while the private LLC configuration suffers performance loss of approx. 10%. On the other hand, in `water.nsq`, where almost all pages are private, DELTA's performance is equivalent to that of the private LLC configuration, achieving a speed-up of 6% over S-NUCA.

The results in Table V indicate that several benchmarks have a low amount of private pages as opposed to private blocks. On reason for this is the existence of shared data, such as boundary elements in structured grid simulations, in pages that are mostly private. In general, this will lead to less locality-optimized cache access for these benchmarks. Due to the additional costs associated with distant memory accesses (both in DRAM and in caches), modern HPC software development encourages programming styles that result in higher number of private pages. This is designed to ensure threads operate on local memory thereby reducing the amount of shared pages [30]. Moreover, an important trend in algorithm design are Communication-avoiding Algorithms (CAA) [31] which, attempt to reduce sharing. Hence, we expect the architecture of DELTA to achieve even better performance on modern multi-threaded workloads with a considerable amount of private data in comparison to S-NUCA. Detailed modeling and optimization of DELTA for emerging multithreaded workloads is left for future research.

D. Frequency of reconfiguration

In order to understand the impact of the frequency by which cache allocations are computed, we simulate a cache partitioning solution that uses an ideal implementation of Lookahead for computing allocations with zero overheads (see Section III-A for details about the ideal centralized implementation) at two cache allocation frequencies (1 ms and 100 ms), on the baseline system. Figure 13 shows the impact of allocation frequency on performance of five different workload mixes each comprising 16 SPEC CPU2006 benchmarks (see Table IV). The results demonstrate that while frequent allocations do not benefit all workloads, they do provide the opportunity to improve performance for several of the workloads considered, because of better adaptation to phase changes.

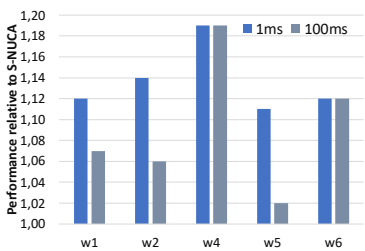


Fig. 13: Impact of frequency of reconfigurations on a 16-core CMP.

E. Overheads

We analyze the different sources of overheads in the centralized scheme and DELTA.

1) Computational overheads: The worst-case time complexity of the Lookahead algorithm is $O(N \times W^2)$ where N is number of cores and W is number of ways. We can consider the algorithm to have cubic complexity, since the number of ways needs to be at least as many as the number of cores (for way-partitioning). The best case complexity is $O(N \times W)$, i.e. quadratic. Peekahead, which considers only the points of the miss rate on a convex hull, has a complexity of $O(N \times W)$, in the best/average case and $O(N \times W^2)$ in the worst-case. The time to compute cache allocations for different core counts using Lookahead and Peekahead, with 16 ways per core, is presented in Table VI.

The Lookahead algorithm takes 5.32 ms on average to compute allocations for a CMP with 16 cores (16-tile CMP with each bank containing 16 ways). Peekahead takes 0.89 ms on average for the same scenario. For larger core counts the overhead is even larger. Note that the data presented in the table do not take into account the additional computations needed to perform locality-aware data placement, which for large core counts has been shown to exceed capacity allocation overheads [32].

For DELTA the complexity can be attributed to the inter- and intra-bank allocation algorithm. The pain and gain computation step takes constant time, i.e. $O(1)$ complexity. The inter/intra bank allocation algorithm requires finding the core with the MIN and MAX gain/pain values. This operation is similar to finding the min. and max. in an unsorted array. The simplicity of the DELTA reconfiguration algorithms enables a hardware implementation with low overheads. Even if the algorithms were to be implemented in software, the overhead of DELTA's inter- and intra-bank allocation algorithms assuming a 64-core CMP would be 0.015 ms and 0.007 ms, three orders of magnitude lower than state-of-the-art.

2) Message overheads: We calculate the number of additional messages sent in the worst-case at each reconfiguration interval (assuming $i_{inter} = 1ms$ and $i_{intra} = 0.1ms$) in a 16-core CMP. For the centralized scheme the total number of messages is $2 \times N$, where N is the number of cores in the system, and this results in $16 \times 2 = 32$ additional

Cores	2	4	8	16	32	64
Lookahead	0.02	0.05	0.46	5.32	73.07	1230
Peekahead	0.03	0.07	0.23	0.89	3.34	13.12

TABLE VI: Overhead for Lookahead and Peekahead in ms per invocation.

messages. DELTA however needs $2 \times N$ messages for intra-bank allocation and $N \times 10 \times 2$ for inter-bank allocation which results in a total of 352 messages. However, the number of messages on the NoC pertaining to L2 misses alone, during the same interval, is 320K on average. This indicates that the overhead in terms of additional messages for DELTA is marginal ($\sim$0.1%), even in the worst-case.

3) Invalidation overheads: Invalidations are needed when remapping addresses to LLC locations regardless of whether the allocation algorithm is centralized or distributed. Invalidation overheads can be primarily attributed to two causes: the overheads of performing the invalidations and how often/much data need to be invalidated. We address the first by performing bulk invalidations (see Section II-C). We mitigate the second by only requiring invalidations for inter-bank reconfigurations. Intra-bank reconfigurations do not lead to invalidations except when a retreat is triggered in rare cases (see Section II-D).

V. Related work

Cache Partitioning: Several solutions have been proposed in literature for cache resource partitioning. Way partitioning is the most popular and it is implemented in commodity systems [2], [19], [33], [34]. It is a simple technique that works by limiting which ways a core can insert into. The major limitation is that it requires cache associativity to scale with the number of cores, which is not easily done [35]. Set partitioning is another approach, which can be implemented with hardware or software support [8], [9], [23], [36]. Hardware based schemes require flexible indexing of the cache. Software schemes use page coloring and rely on OS support for partitioning. Page coloring, however, cannot support superpages and incurs high overhead for reconfiguration. The aforementioned solutions also have the drawback of only supporting a limited number of coarse-grained partitions.

Fine-grained partitioning solutions can be broadly classified in three categories, i) hybrid techniques [10], ii) clustering techniques [11], [12] and iii) replacement-based techniques [13]–[16]. Hybrid techniques like SWAP, combine set and way partitioning in order to get more fine-grained partitions. Clustering techniques like KPart, group applications into clusters and then assigns clusters to way partitions, to emulate fine-grained partitioning. The replacement-based techniques adapt the cache replacement policy to enable fine-grained partitions with different sizes. However, in the context of tile-based CMPs, these approaches leave room for further improvement since they do not take locality into account.

A few proposals performs locality-aware placement for tiled CMPs [4], [18]. CloudCache [18] uses virtual private cache partitions that span across banks and performs locality-aware placement of the partitions. The drawback of this proposal is that it uses N-chance spilling [37] on evictions and requires costly broadcasts. Jigsaw [4] lets software define *shares* and then maps data to them by assigning a share *id* to every page in the application. Allocation and enforcement is done at share granularity instead of application/core granularity as in prior proposals. The proposal relies heavily on software support. Furthermore, in the aforementioned solutions the allocation decision is made by a central hardware or software component which limits scalability.

To lower the overhead of a central component, XChange [38] uses a market-based approach where some of the computations for multi-resource management are done in each core/bank. However when used for partitioning a single resource, XChange will result in an equal partitioning. Moreover the scheme is based on a shared L2 cache structure and thus does not enable locality-aware placement, unlike DELTA.

Non-Uniform Cache Access (NUCA): Efficient usage of NUCA caches has been an extensively researched topic [24], [39]–[45]. The simplest approach, Static NUCA (S-NUCA), spreads the data over all cache banks with a fixed mapping and exposes variable access latencies.

D-NUCA schemes try to combine the best from private and shared cache designs, where private designs have isolation but suffers from under-utilization and shared designs have dynamic utilization but suffers from long on-chip latencies and interference.

A few proposals, like DELTA, try to minimize long on-chip distance. They mostly focus on multi-threaded applications and achieve this by replication and placement, where frequently used lines are copied and placed in the nearest cache bank [42]–[44]. Spilling has been proposed as a way to overcome the problem of underutilized private caches. It works by inserting a copy of a line in another cache bank before it is evicting from the cache hierarchy [37], [46].

Both spilling and replication have two problems. Firstly, both operations decrease the capacity of the cache, where a tradeoff is made between latency and capacity. Secondly, costly directory lookups are needed in order to find data. In our proposal, we avoid both these drawbacks since we place data in a locality-aware way without replicating and do not need a directory to find the data.

Some D-NUCA proposals implement a different partitioning strategy, where separation is done between shared and private regions instead of applications [47]–[50]. Elastic Cooperative Caching (ECC) [50] uses a distributed approach to divide the cache bank between shared/private using way partitioning. The scheme also uses spilling to extend capacity to other banks and therefore inherits the drawbacks of spilling. In contrast to this work we enforce strict per application partitions that can span multiple banks.

R-NUCA [24] classifies accesses into three categories (instructions, private data, shared data) and uses static rules for placement and replication for each type. The drawbacks of the scheme is that it uses a static placement scheme for the different classes not taking dynamic nor application specific behaviour into account, which DELTA does. Note that, compared to DELTA, none of the aforementioned techniques give strict interference protection for data.

Coherence framework: CDR [51] reduces the scope of cache coherence from global to VM-, application-, or page-level to enable shared memory between servers or to minimize on-chip distances in a manycore. Unlike DELTA it does not

provide strict cache partitioning. Furthermore, creation of sharer domains does not take the cache requirements of each application into account but instead allows the different threads of the same application to form a domain.

To the best of our knowledge, DELTA is the first distributed solution for fine-grained and locality-aware cache partitioning which permits permits hardware implementation and scales to many-core architectures.

VI. Conclusions

We present DELTA, a fully distributed and locality-aware partitioning solution for tile-based CMPs. The solution is scalable through its novel challenge-based allocation algorithm, which allocates cache capacity in a distributed way based on the performance gain of each application.

We show that the distributed algorithm has low computational overhead which permits hardware implementation and enables frequent reconfiguration. The allocation algorithm is supported by a flexible enforcement mechanism that enables locality-aware placement. Our evaluation demonstrates that the distributed partitioning solution performs close to an ideal centralized solution and scales to large core counts.

Acknowledgment

This research has been funded by the European Research Council, under the MECCA project, contract number ERC-2013-AdG 340328 and the Swedish Research Council (VR), under the ACE project. The simulations were performed on resources at Chalmers Centre for Computational Science and Engineering (C3SE) provided by the Swedish National Infrastructure for Computing (SNIC).

References

[1] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *Proc. MICRO-49*, 2006.
[2] L. R. Hsu *et al.*, "Communist, Utilitarian, and Capitalist Cache Policies on CMPs: Caches as a Shared Resource," in *PACT-15*, 2006.
[3] D. Thiebaut, H. S. Stone, and J. L. Wolf, "Improving disk cache hit-ratios through cache partitioning," *IEEE Trans. on Comp.*, 1992.
[4] N. Beckmann and D. Sanchez, "Jigsaw: Scalable software-defined caches," in *Proc. PACT-22*, 2013.
[5] R. Riesen *et al.*, "Designing and implementing lightweight kernels for capability computing," *Concurrency and Computation: Practice and Experience*, 2009.
[6] T. Hoefler, T. Schneider, and A. Lumsdaine, "Characterizing the influence of system noise on large-scale applications by simulation," in *Proc. SC'10*, 2010.
[7] T. Jones, "Linux kernel co-scheduling for bulk synchronous parallel applications," in *Proceedings of the 1st International Workshop on Runtime and Operating Systems for Supercomputers*, ser. ROSS '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 57–64.
[8] J. Lin *et al.*, "Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems," in *Proc. HPCA-14*, 2008.
[9] S. Cho and L. Jin, "Managing distributed, shared l2 caches through os-level page allocation," in *Proc. MICRO-39*, 2006.
[10] X. Wang *et al.*, "SWAP: Effective Fine-Grain Management of Shared Last-Level Caches with Minimum Hardware Support," in *Proc. HPCA-23*, 2017.
[11] H. Cook *et al.*, "A hardware evaluation of cache partitioning to improve utilization and energy-efficiency while preserving responsiveness," in *Proc. ISCA-40*, 2013.
[12] N. El-Sayed *et al.*, "KPart: A Hybrid Cache Partitioning-Sharing Technique for Commodity Multicores," in *Proc. HPCA-24*, 2018.
[13] G. Kurian, O. Khan, and S. Devadas, "The locality-aware adaptive cache coherence protocol," *ACM SIGARCH Comput. Archit. News*, 2013.
[14] R. Manikantan, K. Rajan, and R. Govindarajan, "Probabilistic shared cache management," in *Proc. ISCA-39*, 2012.
[15] D. Sanchez and C. Kozyrakis, "Vantage: Scalable and Efficient Fine-Grain Cache Partitioning," *Proc. ISCA-38*, 2011.
[16] Y. Xie and G. H. Loh, "Pipp: Promotion/insertion pseudo-partitioning of multi-core shared caches," in *Proc. ISCA-36*, 2009.
[17] R. Iyer, "Cqos: A framework for enabling qos in shared caches of cmp platforms," in *Proc. ICS-18*, 2004.
[18] H. Lee, S. Cho, and B. R. Childers, "CloudCache: Expanding and shrinking private caches," in *Proc. HPCA-17*, 2011.
[19] A. Herdrich *et al.*, "Cache QoS: From concept to reality in the Intel® Xeon® processor E5-2600 v3 product family," in *Proc. HPCA-22*, 2016.
[20] J. Gandhi *et al.*, "Range translations for fast virtual memory," *IEEE Micro*, 2016.
[21] R. Wang and L. Chen, "Futility scaling: High-associativity cache partitioning," in *Proc. MICRO-47*, 2014.
[22] "Intel® 64 and IA-32 Architectures Developer's Manual."
[23] M. Awasthi *et al.*, "Dynamic hardware-assisted software-controlled page placement to manage capacity allocation and sharing within large caches," in *Proc. HPCA-15*, 2009.
[24] N. Hardavellas *et al.*, "Reactive NUCA: Near-Optimal Block Placement and Replication in Distributed Caches," in *Proc. ISCA-36 2009*, 2009.
[25] T. E. Carlson *et al.*, "An evaluation of high-level mechanistic core models," *ACM TACO*, 2014.
[26] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, "Optimizing nuca organizations and wiring alternatives for large caches with cacti 6.0," in *Proc. MICRO-40*, 2007.
[27] T. Sherwood *et al.*, "Automatically characterizing large scale program behavior," in *Proc. ASPLOS-10*, 2002.
[28] S. Eyerman and L. Eeckhout, "System-level performance metrics for multiprogram workloads," *IEEE Micro*, 2008.
[29] C.-K. Luk *et al.*, "Pin: building customized program analysis tools with dynamic instrumentation," in *Acm sigplan notices*, vol. 40, no. 6. ACM, 2005, pp. 190–200.
[30] D. Ott, "Optimizing Applications for NUMA," 2011.
[31] J. Demmel, "Communication avoiding algorithms," in *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, Nov 2012, pp. 1942–2000.
[32] N. Beckmann, "Design and analysis of spatially-partitioned shared caches," Ph.D. dissertation, Massachusetts Institute of Technology, 2015.
[33] S.-H. Yang *et al.*, "Exploiting choice in resizable cache design to optimize deep-submicron processor energy-delay," in *Proc. HPCA-8*, 2002, pp. 151–161.
[34] G. E. Suh, S. Devadas, and L. Rudolph, "A new memory monitoring scheme for memory-aware scheduling and partitioning," in *Proc. HPCA-8*, 2002, pp. 117–128.
[35] D. Sanchez and C. Kozyrakis, "The zcache: Decoupling ways and associativity," in *Proc. MICRO-43*, 2010, pp. 187–198.
[36] D. M. Tullsen and J. A. Brown, "Handling long-latency loads in a simultaneous multithreading processor," in *Proc. MICRO-34*, 2001.
[37] M. K. Qureshi, "Adaptive spill-receive for robust high-performance caching in CMPs," in *Proc. HPCA-15*, 2009.
[38] X. Wang and J. F. Martinez, "XChange: A market-based approach to scalable dynamic multi-resource allocation in multicore architectures," in *Proc. HPCA-21*, 2015.
[39] S. Srikantaiah *et al.*, "MorphCache: A Reconfigurable Adaptive Multi-level Cache hierarchy," in *Proc. HPCA-17*, 2011.
[40] A. Sembrant, E. Hagersten, and D. Black-Schaffer, "The Direct-to-Data (D2D) Cache: Navigating the cache hierarchy with a single lookup," in *Proc. ISCA-42*, 2014.
[41] ——, "A Split Cache Hierarchy for Enabling Data-Oriented Optimizations," in *Proc. HPCA-23*, 2017.
[42] M. Zhang *et al.*, "Victim Replication: Maximizing Capacity while Hiding Wire Delay in Tiled Chip Multiprocessors," in *Proc. ISCA-32*, 2005.
[43] B. M. Beckmann, M. R. Marty, and D. A. Wood, "ASR: Adaptive selective replication for CMP caches," in *Proc. MICRO-39*, 2006.
[44] P. A. Tsai, N. Beckmann, and D. Sanchez, "Nexus: A New Approach to Replication in Distributed Shared Caches," in *Proc. PACT-26*, 2017.
[45] Q. Shi *et al.*, "LDAC: Locality-Aware Data Access Control for Large-Scale Multicore Cache Hierarchies," *ACM TACO*, 2016.
[46] J. Chang and G. S. Sohi, "Cooperative Caching for Chip Multiprocessors," in *Proc. ISCA-33*, 2006.
[47] H. Dybdahl and P. Stenstrom, "An adaptive shared/private nuca cache partitioning scheme for chip multiprocessors," in *Proc. HPCA-13*, 2007.
[48] L. Zhao *et al.*, "Towards hybrid last level caches for chip-multiprocessors," *ACM SIGARCH Comput. Archit. News*, 2008.
[49] J. Merino, V. Puente, and J. A. Gregorio, "ESP-NUCA: A low-cost adaptive Non-Uniform Cache Architecture," in *Proc. HPCA-6*, 2010.
[50] E. Herrero *et al.*, "Elastic cooperative caching," in *Proc. ISCA-37*, 2010.
[51] Y. Fu, T. M. Nguyen, and D. Wentzlaff, "Coherence domain restriction on large scale systems," in *Proc. MICRO-48*, 2015.

Paper II

CBP: Coordinated management of cache partitioning, bandwidth partitioning and prefetch throttling

N. Ramhöj Holtryd, M. Manivannan, P. Stenström and M. Pericàs

Under review.

CBP: Coordinated management of cache partitioning, bandwidth partitioning and prefetch throttling

Nadja Ramhöj Holtryd, Madhavan Manivannan, Per Stenström, Miquel Pericàs
Department of Computer Science and Engineering
Chalmers University of Technology
Göteborg, Sweden
Email:{holtryd,madhavan,per.stenstrom,miquelp}@chalmers.se

ABSTRACT

Reducing the average memory access time is crucial for improving the performance of applications running on multi-core architectures. With workload consolidation this becomes increasingly challenging due to shared resource contention. Techniques for partitioning of shared resources - cache and bandwidth - and prefetching throttling have been proposed to mitigate contention and reduce the average memory access time. However, existing proposals only employ a single or a subset of these techniques and are therefore not able to exploit the full potential of coordinated management of cache, bandwidth and prefetching. Our characterization results show that application performance, in several cases, is sensitive to prefetching, cache and bandwidth allocation. Furthermore, the results show that managing these together provides higher performance potential during workload consolidation as it enables more resource trade-offs. In this paper, we propose CBP a coordination mechanism for dynamically managing prefetching throttling, cache and bandwidth partitioning, in order to reduce average memory access time and improve performance. CBP works by employing individual resource managers to determine the appropriate setting for each resource and a coordinating mechanism in order to enable inter-resource trade-offs. Our evaluation on a 16-core CMP shows that CBP, on average, improves performance by 11% compared to the state-of-the-art technique that manages cache partitioning and prefetching and by 50% compared to the baseline without cache partitioning, bandwidth partitioning and prefetch throttling.

1 INTRODUCTION

Memory access time has a significant impact on application performance. Effective utilization of the memory system is therefore necessary. Typically, resources in the memory system (e.g., last-level cache (LLC) and off-chip memory bandwidth) are shared among multiple cores as they help in improving resource utilization during workload consolidation. However, sharing can detrimentally impact average memory access time and performance due to resource contention. Prior works have proposed partitioning of shared resources – cache [10, 15, 19, 26, 28] and bandwidth [13, 17, 24] – and prefetching [11] to mitigate contention, reduce or hide memory access time and improve performance.

Recent works have proposed combining cache and bandwidth partitioning [4, 23, 27], prefetching and cache partitioning [31, 33] and bandwidth partitioning and prefetching [8, 18] to provide additional performance gains. The key insight from these papers is that coordinated management of two techniques is more advantageous than considering each in isolation because of the trade-offs that are made possible. However, no study so far has considered combining all three techniques. The goal of this paper is to do so.

Coordinated management of cache partitioning, bandwidth partitioning and prefetch throttling provides the following advantages. Firstly, it makes it possible to address more applications and cover a broader range of workloads, as shown in our in-depth performance characterization (see Section 2). The results show that 90% of the applications in the SPEC CPU2006 suite have performance sensitivity (over 10% change in IPC) to at least one of the techniques, and 70% are also sensitive to multiple techniques. Secondly, managing these techniques jointly opens up the opportunity to new and improved trade-offs. There are synergistic interactions between the techniques and these cannot be realized if cache partitioning, bandwidth partitioning and prefetch throttling are not jointly managed.

As an example, consider the simple case of a workload comprising of two applications. The first application, *lbm*, is sensitive to bandwidth and prefetching while the second, *xalancbmk*, is sensitive to cache size and has lower performance when prefetching is enabled. The best solution when managing all three techniques is to give xalancbmk a large cache allocation, small bandwidth allocation and disable the prefetcher, while giving lbm a large bandwidth allocation, small cache allocation while keeping the prefetcher active. Figure 1 shows the performance from coordinated management of all three techniques (*cache+bw+pref*) compared to managing a subset of the techniques. The results show that the solution that manages all three techniques is better than others that manage two of the techniques, and leads to an additional performance gain of 15%. The main challenge of coordinately managing all three techniques is the complexity of evaluating all possible allocations

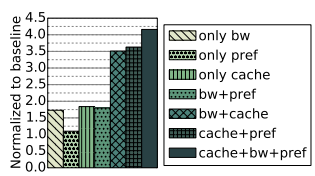


Figure 1: Workload with lbm and xalancbmk. Total bandwidth 16GB/s, total cache size 2MB. Executing applications for 1B instructions, more details in Section 4. Settings: lbm-prefethingactive, 12GB/s, xalancbmk- prefetcher inactive, 4GB/s, determined from characterization (Section 2). Cache partition sizes are decided dynamically.

dynamically and determining the best possible allocation while exploiting the large number of possible trade-offs.

Guided by our characterization results, we propose CBP, a coordinated mechanism for dynamically managing cache partitioning, bandwidth partitioning and prefetch throttling for multi-programmed workloads. CBP consists of three local controllers, one for each resource that together with a coordination mechanism manages and allocates the resources. With CBP, the three techniques are dynamically tuned in an iterative fashion. First, cache space is allocated since avoiding a memory access altogether is better than reducing their latency. As a next step, bandwidth is allocated taking into consideration the impact due to cache allocation. Lastly, the prefetch setting is determined by testing the impact of prefetching on performance for the current allocation of bandwidth and cache. The prefetcher performance influences the next reallocation of cache space and bandwidth. The feedback mechanism between the different techniques dynamically adapts the allocations in order to reach a good configuration depending on the characteristics of the individual applications in the workload. Our approach of combining local controllers with a feedback mechanism reduces the complexity.

In summary, we make the following contributions:

(a) We present an in-depth characterization of the performance impact of cache, bandwidth and prefetching on the entire SPEC CPU2006 suite. Our characterisation results provide several insights: i) a majority of the applications (over 90%) are sensitive to one or multiple techniques, ii) managing cache, bandwidth and prefetch, opens up opportunities for exploiting more trade-offs and improving performance for consolidated workloads, and iii) managing cache, bandwidth and prefetch jointly has the potential to outperform combinations of two of the techniques.

(b) We propose CBP, a mechanism to dynamically manage the three resources in coordination. The solution is based on simple heuristics in order to sidestep the complexity associated with evaluating all possible configurations and choosing the most efficient configuration. CBP works by employing individual resource managers to determine the appropriate setting for each resource and a coordinating mechanism to enable inter-resource trade-offs.

(c) We evaluate our solution with multi-programmed workloads on a 16-core tiled CMP. CBP improves performance by up to 36% (geom. mean 11%) compared to the state-of-the-art technique that manages cache partitioning and prefetching in a coordinated manner and by up to 86% (geom. mean 50%) compared to an unpartitioned S-NUCA without cache partitioning, bandwidth partitioning and prefetching.

The rest of the paper is organized as follows. Section 2 motivates the need for a coordinated approach using cache partitioning, bandwidth partitioning and prefetch throttling. Section 3 describes our proposed solution in detail. We then discuss the methodology in Section 4 and Section 5 presents the evaluation of the proposal. We provide an overview of related work in Section 6 and conclude in Section 7.

2 CHARACTERIZATION

In order to motivate the need for coordinated management of cache partitioning, bandwidth partitioning and prefetch throttling, we perform a detailed characterization study of applications in the SPEC2006 CPU suite. The aim of this study is to: i) characterize applications to determine the extent to which they are performance sensitive to cache, bandwidth and prefetch settings, ii) understand the different resource interactions, their impact on performance and the inter-resource trade-offs that are possible, and iii) demonstrate the performance potential of coordinated management of all three resources over a subset of resources.

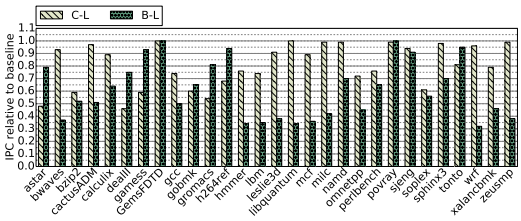
2.1 Sensitivity to cache, bandwidth and prefetch settings

To understand the sensitivity of applications to cache, bandwidth and prefetch settings we model a system consisting of one out-of-order core with a 3-level cache hierarchy using the Sniper simulator [5]. Details about the methodology are provided in Section 4. For this experiment, the baseline LLC and bandwidth allocation is 512kB and 4GB/s, respectively. We run the application in steady-state for 1B instructions and use IPC as a measure of performance.

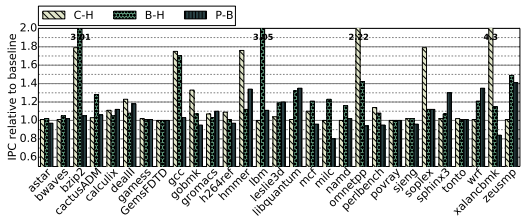
Figure 2 shows the performance impact of changing the cache allocation, bandwidth allocation and enabling prefetching normalized to the baseline allocation without prefetching. Note that we only change the setting for one resource at a time. In Figure 2a, C-L and B-L represent low allocation settings where the cache allocation is decreased to 128kB and the bandwidth allocation is decreased to 1GB/s, respectively, while prefetching is disabled. Similarly, in Figure 2b, C-H and B-H represent high allocation settings where the cache allocation is increased to 2MB and the bandwidth allocation is increased to 16GB/s, while prefetching is disabled. Finally, P-B represents the setting where prefetching is enabled with the baseline cache and bandwidth allocation. We classify applications as *performance sensitive* to a specific resource if the modified allocation results in a 10% deviation from the baseline IPC. We refer to applications that are performance sensitive to change in cache allocation as cache sensitive (CS), sensitive to change in bandwidth allocation as bandwidth sensitive (BS) and sensitive to prefetch throttling as prefetch sensitive (PS).

The sensitivity results for cache size show that nearly 60% of the applications (17 out of 29) are sensitive to changes in cache allocation. The extent to which applications are performance sensitive varies greatly with a performance increase of up to 4x in some cases. Furthermore, a larger number of applications are sensitive in the low allocation setting in comparison to high allocation setting (17 compared to 11). The sensitivity results for bandwidth allocation also shows a similar trend, as more applications are sensitive in the low allocation setting (23 compared to 15). Also, the extent of performance sensitivity varies greatly with an increase of up to 3x. The sensitivity results for prefetch throttling indicate that nearly 38% of the applications (11 out of 29) are sensitive to prefetching and experience a speedup. However, there are some applications that experience a slowdown due to prefetching. In summary, we make the following observation:

OBSERVATION 1. *In SPEC CPU2006 suite 90% of the applications are sensitive to one resource, while 70% are sensitive to multiple resources and the extent of sensitivity varies greatly.*



(a) Slowdown when decreasing the cache size to 128kB (C-L), and slowdown when decreasing the bandwidth allocation to 1GB/s (B-L) in comparison to the baseline allocation, with prefetching disabled.



(b) Performance improvement from increasing cache allocation to 2MB (C-H), and increasing bandwidth allocation to 16GB/s (B-H) in comparison to the baseline allocation, with prefetching disabled. Performance improvement from enabling prefetching for the baseline allocation (P-B)

Figure 2: Performance impact of changing cache size, bandwidth allocation and prefetcher setting. There are 6 CS-BS-PS applications, 8 CS-BS, 6 BS-PS, 3 CS, 3 BS and 3 applications are insensitive (I) to all three techniques.

2.2 Inter-resource interactions and trade-offs

Next, we investigate inter-resource interactions and trade-offs that are enabled when jointly managing cache, bandwidth and prefetching. We focus on intra-application resource interaction initially. There are four possible types of interactions within an application: cache-bandwidth-prefetch, bandwidth - prefetch, cache - prefetch and cache - bandwidth.

Regarding the cache-bandwidth-prefetch trade-off, we want to find out how the performance impact from prefetching varies with the allocation of cache and bandwidth. Figure 3 shows the performance impact of prefetching for three different cache/bandwidth settings normalized to the respective baseline setting without prefetching. The cache and bandwidth setting for an application in a low allocation scenario (P-L) is 128kB and 1GB/s, the baseline allocation scenario (P-B) setting is 512kB and 4GB/s while the high allocation scenario (P-H) setting is 2MB and 16GB/s.

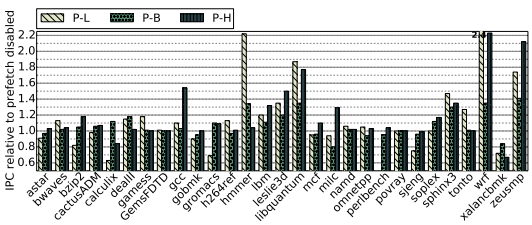


Figure 3: Performance impact of enabling prefetching relative to allocation of cache and bandwidth, for allocation settings; L:128kB,1GB/s B:512kB,4GB/s H:2MB,16GB/s

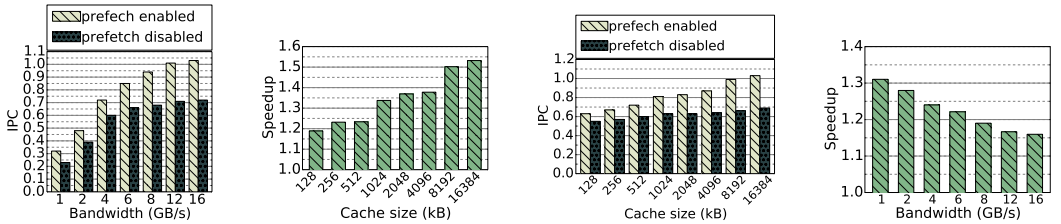
For some applications, lower bandwidth and cache allocation leads to higher sensitivity for prefetching as seen in *hmmer*. This is because avoiding a miss altogether, as a consequence of accurate prefetching, can have a larger impact in low allocation settings where the bandwidth is scarce and the memory queuing delays tend to be longer. Also, there are applications like *gcc* which experience higher prefetch sensitivity with a larger cache and bandwidth allocation. The results indicate that applications tend to be prefetch sensitive in some settings and prefetch insensitive in others. We make the following observation:

OBSERVATION 2. *Allocation of cache and bandwidth influences prefetch sensitivity. Furthermore, applications tend to be prefetch sensitive in some settings and prefetch insensitive in others.*

In the interest of space we use *leslie3d* as a representative example to illustrate the other pairwise resource interactions and trade-offs, since it is sensitive to all three techniques. Note that the baseline setting will be used for the resources unless specified otherwise. The bandwidth - prefetch interaction manifests in two different ways. Firstly, prefetching typically increases the number of memory accesses and this in turn increases the bandwidth pressure. In the case of *leslie3d* prefetching results in a 15% increase in the number of memory requests in comparison to the baseline. Note that prefetch misses, i.e. prefetched blocks that are evicted before use, can result in a further increase in pressure on the memory bandwidth. Secondly, the performance improvement from prefetching can be influenced by the bandwidth allocation. The results in Figure 4a show the performance for different bandwidth allocation with and without prefetching. We make the following observation:

OBSERVATION 3. *A larger bandwidth allocation can compensate for increased bandwidth demands, due to inaccurate prefetches, leading to increased performance with prefetching.*

The cache - prefetch interaction also manifests in two main ways. Firstly, the performance loss from reduced cache allocation can be offset if prefetching is effective. Figure 4c shows the IPC for different cache allocations with and without prefetching. The results show that the performance of 128kB allocation with prefetching is better than 512kB allocation without prefetching. Secondly, larger cache sizes (if it is used efficiently) can lead to higher speedup from prefetching. Figure 4b shows the performance improvement from prefetching with different cache allocation normalised to the respective cache allocation without prefetching. The results show that prefetching is effective with lower cache allocation and that



(a) IPC with and without prefetching for different bandwidth allocation.

(b) Performance improvement from prefetching depending on cache allocation.

(c) IPC with and without prefetching for different cache allocation.

(d) Performance improvement from increasing cache allocation to 2MB depending on bandwidth allocation.

Figure 4: leslie3d - example of interactions and its impact on performance within a single application.

its effectiveness can increase with additional allocation. The reason for this behaviour is that a larger cache reduces the number of memory accesses which has the same effect as increasing the available bandwidth, i.e. a lower queuing delay, which is more forgiving when there is an increase in memory accesses caused by inaccurate prefetches (in leslie3d there is a 15% increase in dram accesses caused by prefetching). These results lead to the following observation:

OBSERVATION 4. *A trade-off can be made between either increasing cache size or enabling prefetching, leading to the same performance, for applications which are performance sensitive to both cache and prefetching.*

As for the cache - bandwidth interaction, a lower bandwidth allocation can result in a larger sensitivity to cache size. Figure 4d shows the performance improvement from increasing the cache allocation from 512kB to 2MB with different bandwidth allocation settings. The results show that performance improvement from additional cache allocation is much higher in low bandwidth allocation settings (see the result for 1GB/s bandwidth allocation). This is because the average cost of a miss is much higher in the case of lower bandwidth allocation. The results also show that a large cache allocation can reduce the performance sensitivity to bandwidth allocation (see the result for 16GB bandwidth allocation). These results lead to the following observation:

OBSERVATION 5. *A trade-off can be made between either increased cache space or increased bandwidth allocation, for applications which are performance sensitive to both cache and bandwidth.*

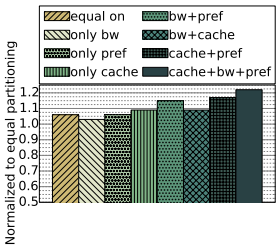
We now describe how the observed intra-application interactions and trade-offs can be leveraged in the inter-application setting for multi-programmed workloads. Let us revisit the example of running a simple workload comprising two application (*lbm* and *xalancbmk*) on a dual-core system with 2MB LLC capacity and 16GB/s bandwidth, discussed in Figure 1. For achieving the best aggregate performance we expect xalancbmk to get the majority of the cache (nearly 1.75MB), while lbm is given a smaller cache allocation of 256KB. For bandwidth, we would expect lbm to have a large allocation (12GB/s) of the available bandwidth and xalancbmk to get a smaller allocation (4GB/s). This is reflected in Observation 5 about the trade-off between cache and bandwidth where we would prioritize the application that shows the highest sensitivity for the resource. Furthermore, as reflected in Observation 2, we expect prefetching to be more effective for lbm since it has a large allocation of bandwidth. In the case of xalancbmk, prefetching leads to lower performance regardless of the allocation of cache and bandwidth.

2.3 Potential for coordinated management

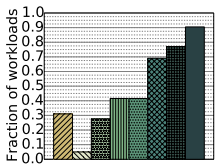
In order to show the potential for coordinated management of cache, bandwidth and prefetch we run 640 randomly generated workloads each comprising 4 SPEC 2006 CPU applications. We compare the performance of jointly managing all the three resources to other resource managers that only manage a subset of these resources. For this experiment the baseline allocation of cache and bandwidth for each application is 512kB and 4GB/s. We use an exhaustive search algorithm to find the best static configuration (over 1B instructions) for the different resources when running each workload. Figure 5a shows average (geometric mean) performance with different resource managers normalized to the baseline settings without prefetching. *equal on*, depicts the performance when prefetching is enabled for all applications and improves performance by 6% while *only pref*, depicts the performance when prefetching is selectively activated and improves performance by 9%. *cache+bw+pref* results show that coordinately managing cache partitioning, bandwidth partitioning and prefetch throttling improves performance by 5% compared to the best combination of two techniques (22% compared to 17%).

Figure 5b shows the number of workloads (among the 640 workloads considered) that experience a performance gain of at least 10% using the different resource managers discussed previously. The results show that 90% (597) of the workloads are sensitive to the resource manager that jointly manages all three techniques. A smaller fraction of the workloads are sensitive to resource managers that manage a subset of these techniques (77% are sensitive to *cache+pref* resource manager and 69% are sensitive to the *cache+bw* resource manager).

In summary, the results from the characterization study demonstrate that around 90% of applications in the SPEC 2006 CPU suite are sensitive to different resources and that coordinately managing them opens up new possibilities for improving performance



(a) Performance potential of using different resource managers.



(b) Fraction, of the workloads, with at least 10% performance improvement.

Figure 5: Potential for coordinated management measured using 640 random workloads of 4 SPEC CPU2006 applications. Performance is obtained using an exhaustive search algorithm that evaluates bandwidth settings (2GB/s, 4GB/s, 6GB/s), cache settings (256kB, 512kB, 1024kB) and prefetching settings (active/inactive), in conjunction, to determine the resource allocation for each application in the workload that maximizes aggregate performance.

by trading resource allocations. Furthermore, jointly managing these resources has the potential to cover a broader range of workloads and outperform resource managers that manage a subset of resources. In the next section we will discuss how the proposed resource manager, CBP, determines cache, bandwidth and prefetch settings for different applications in a workload.

3 CBP RESOURCE MANAGER

Section 3.1 provides an overview of the CBP resource manager. Section 3.2 discusses the individual resource controllers while Section 3.3 describes how the coordination mechanism ties the local resource controllers together. Finally, the implementation and overhead of the proposed mechanism is discussed in Section 3.4.

3.1 Overview

CBP is a coordinated mechanism for dynamically managing cache partitioning, bandwidth partitioning and prefetch throttling. The design consists of one local controller for each of the three techniques, and a coordination mechanism, as shown in Figure 6.

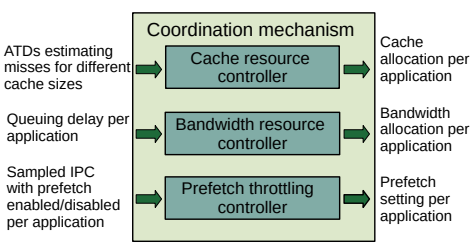


Figure 6: Overview of CBP resource manager.

The cache allocation controller estimates the number of misses for different cache sizes using auxiliary tag directories (ATDs) [26] and uses this as input for determining cache allocation. The cache allocation per application is determined such that it reduces the aggregate number of cache misses for the entire workload. The bandwidth allocation controller uses memory request queuing delay experienced by the applications as input and allocates the available bandwidth in proportion to the delay. The bandwidth allocation controller, assigns a larger allocation of the available bandwidth to applications that experience longer queuing delay, and a comparatively lower allocation to those that experience shorter queuing delays. Lastly, the prefetch controller samples IPC with and without prefetching to determine whether the prefetcher should be enabled/disabled for each application.

The three techniques are dynamically tuned using the coordination mechanism in an iterative manner such that the local controller takes into consideration the decisions taken by the other controllers.

3.2 Local resource allocation controllers

A partitioning solution for shared resources like cache and bandwidth typically comprises two components: allocation policy, that determines how a resource is divided among multiple co-running applications, and an enforcement mechanism, that enforces the partitioning decision. Similarly, prefetch throttling involves a policy to determine the best prefetch setting to use and a mechanism implemented in hardware to enforce the setting. In the context of CBP, the policy component is of particular interest because it enables inter-resource trade-offs. We discuss the allocation policy in this section and defer the details of the enforcement mechanism to Section 3.4.

3.2.1 Cache partitioning. The cache allocation controller uses the Lookahead algorithm [26], to determine cache allocation. In a nutshell, the algorithm computes the utility for each application where utility is the measure of how many additional misses can be reduced with allocation of cache ways. It then computes the number of ways that maximizes the utility for each application (while ensuring this is less than the total number of ways available for allocation). Finally, it compares the utility values for the different applications, determines the application that has the highest utility and assigns the pre-computed number of ways that maximizes the utility for that application. The process repeats, with recomputation of utility for each application and reassignment of available cache ways to the application that has the largest utility, until the rest of the available capacity is distributed. The allocation controller relies on sampled ATDs to estimate, based on past behaviour, the number of misses that can be avoided with additional allocation of cache ways for each application. In order to adapt to an inclusive cache hierarchy, we assign a minimum allocation of cache space (*min_ways*) to all the applications before distributing the remaining capacity.

3.2.2 Bandwidth partitioning. We propose a bandwidth allocation algorithm, that partitions bandwidth proportional to the memory queuing delay experienced by each application. The pseudo-code for the proposed bandwidth allocation controller is outlined in Algorithm 1. The controller assigns a minimum bandwidth allocation (*min_bandwidth_allocation*) for each application in order to avoid unfairly giving a very low allocation to applications with a small

Algorithm 1: Bandwidth allocation controller pseudo-code

```
   Input  : A list queuingDelayPerApplication
   Output : A list bandwidthAllocationPerApplication
1  At time period reconfiguration_interval;
2  remainingBandwidth = (totalBandwidth-
     min_bandwidth_allocation*totalNumberOfCores
     totalDelay = 0;
3  for i ← 0 to totalNumberOfCores−1 do
4  |   totalDelay += queuingDelayPerApplication[i];
5  |   bandwidthAllocationPerApplication[i] =
   |     min_bandwidth_allocation;
6  end
7  for i ← 0 to totalNumberOfCores−1 do
8  |   bandwidthAllocationPerApplication[i] +=
   |     (queuingDelayPerApplication[i]/totalDelay) *
   |     remainingBandwidth;
9  end
```

queuing delay. The remaining bandwidth is set for distribution among the applications (see line 2). The algorithm computes the total queuing delay by summing up the individual queuing delays experienced by each application (line 4) while assigning the minimum allocation to each application (line 5). As the next step, the remaining bandwidth is allocated proportionally to the queuing delay experienced by the application (line 7-9). The queuing delay for each application is obtained by measuring the memory access time for requests from each application.

3.2.3 Prefetch throttling. The prefetch throttling policy determines the best prefetcher settings for each application. The pseudo-code for prefetch throttling controller is outlined in Algorithm 2. The algorithm considers two possible settings – prefetcher enabled and prefetcher disabled – but can easily be extended to support other aggressiveness settings as well. The algorithm uses the sampled IPC values, for each application obtained, with different prefetcher setting over a sample period (*prefetch_sampling_period*) as input. The algorithm first computes the speedup from prefetching for each application using the sampled IPC values. If the speedup is below a threshold (*speedup_threshold*) the prefetcher is deactivated for the next prefetch interval (*prefetch_interval*) (line 3-4). If the speedup is above the threshold prefetching is activated for the next prefetch interval (line 6). The prefetch throttling controller is generic enough to support any type of prefetcher.

Algorithm 2: Prefetch throttling controller pseudo-code

```
   Input  : Two lists ipcWithPrefetchingActive,
            ipcWithPrefetchingInactive
   Output : A list prefetchSettingPerCore
1  At time period prefetch_interval;
2  for i ← 0 to totalNumberOfCores−1 do
3  |   if
   |     (ipcWithPrefetchingActive[i]/ipcWithPrefetchingInactive[i])
   |     > speedup_threshold then
4  |   |   prefetchSettingPerCore[i] = 0;
5  |   else
6  |   |   prefetchSettingPerCore[i] = 1;
7  |   end
8  end
```

3.3 Coordination Mechanism

The goal of the coordination mechanism is to ensure that each local controller takes into account the decisions taken by other controllers. This is necessary to exploit the trade-offs outlined earlier in Section 2. There are two essential tasks carried out in order to establish this: i) controller prioritization and ii) inter-controller interaction.

Controller prioritization: Since the decision taken by one controller has the potential to influence those taken by others it is important to establish priority among the different local controllers since that determines the order in which local controllers make allocation decisions. In CBP, the highest priority is given to the cache allocation controller, which first makes the allocation decision. The rationale is that avoiding a memory access is typically more effective, for reducing average memory access time, than lowering the memory access penalty. Next, priority is given to the bandwidth allocation controller since our characterization results show that applications are comparatively more sensitive to bandwidth than to prefetching. The least priority, is given to the prefetch throttling controller. This is because it is important that the prefetcher setting is determined based on the current allocation, of cache and bandwidth, since prefetching can have a negative impact on performance if the bandwidth allocation is insufficient.

Inter-controller interaction: Figure 7 provides an overview of the interactions that happens between the different resource allocation controllers. Firstly, we describe how the bandwidth allocation controller decisions takes into account the decisions made by the cache and the prefetch controller. The bandwidth allocation controller, makes decisions based on the queuing delay of each application which is affected by the number of memory accesses. The cache allocation controller through a larger cache allocation can reduce the number of memory accesses (Interaction #1). This leads to a lower bandwidth allocation for applications that can efficiently use the cache. The prefetch throttling controller, influences bandwidth allocation decision mainly through prefetch misses (prefetched data that is not used) (Interaction #2). This is because prefetch misses lead to more memory requests and potentially a higher queuing delay.

Next, we describe how the prefetch throttling controller takes into account the decisions made by the other controllers. The prefetch throttling controller, makes decisions by sampling the IPC with different prefetcher setting over a specific interval. The sampled IPC values, used to determine the prefetcher setting, reflects the effect of cache and bandwidth allocation decisions made by the respective resource controllers (Interaction #3-4). Finally, we describe how the cache allocation controller is affected by the prefetch throttling controller (Interaction #5). If an application benefits from prefetching this reflects on the hit and miss count values

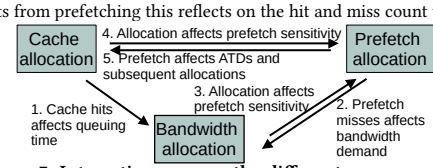


Figure 7: Interactions among the different resource allocation controllers in CBP .

monitored in the ATDs. Since the cache allocation, is computed based on the counter values observed in the ATD, this end up affecting the subsequent cache allocation decision, resulting in a smaller cache allocation for prefetch sensitive applications.

Putting it all together: We discuss the timeline showing when the different resource allocation controllers are invoked and how they interact with each other, in an iterative manner, using an example illustrated in Figure 8. The three resource controllers, are invoked after a specific interval that we refer to as the *reconfiguration_interval* in the sequence shown in the figure. First cache and bandwidth are equally partitioned among all applications at time 0, since information about misses and queuing delay is initially unavailable, as shown in Step 0. This is followed by sampling the IPC of applications with different prefetch settings for a specific interval (twice the *prefetch_sampling_period*) as shown in Step 1. Based on the sampled IPCs the prefetch throttling controller determines the appropriate prefetcher setting for each prefetcher for the current *reconfiguration_interval*. Cache and bandwidth allocation controllers are again invoked after the *reconfiguration_interval*, as shown in Step 2. A cache allocation decision, for the next interval is influenced by the number of hits and misses observed in the previous interval. The ATD values will be halved after each reconfiguration, in order to be sensitive to changes in the last time interval while the per application queuing delays are accumulated with those from the previous interval. The bandwidth allocation decision shown in Step3, is influenced both by the cache allocation and prefetcher setting in the previous interval as discussed previously. Finally, the prefetcher throttling controller shown in Step 4 is influenced by the new cache and bandwidth allocation. The interactions among the different resource allocation controllers, take place over multiple iterations, and is the key to finding an effective solution.

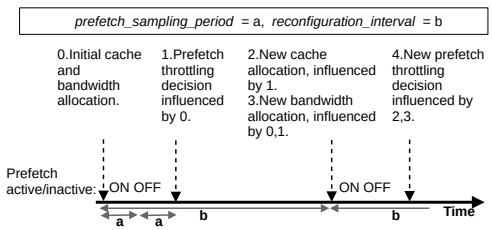


Figure 8: Timing and interactions of CBP resource manager.

3.4 Implementation

The computational overhead of CBP resource management is low since the design uses heuristics to guide the allocation decisions instead of exhaustively evaluating the different possible allocations.

The cache allocation controller needs hardware support in order to estimate the number of misses with different cache sizes. We use sampled ATDs [26] as discussed previously to compute the effect of different cache allocations on the misses. When enforcing cache partitioning there is an overhead associated with invalidations due to reconfiguration decisions. This is modelled faithfully by invalidating the addresses and re-fetching them when accessed, this includes the latency and impact on bandwidth from accessing memory. We have used the enforcement mechanism proposed by Holtryd et al. [12] since it is suitable for a modern tile-based CMP and is both fine-grained and locality aware. The enforcement mechanism uses per-core Cache Bank Tables (CBTs), where mappings between addresses and banks are recorded. When a request needs to access the LLC, the CBT is used to identify the cache bank that the address is mapped to. Inside each bank, way partitioning hardware divides the capacity. The enforcement results in a partition granularity of 32kB on our system, see Section 4. We incur hardware cost for implementing ATDs and cache partition enforcement [12, 26].

The bandwidth partition enforcement is done in likeness with Intel Memory Bandwidth Allocation (MBA) technology [1, 2] which is commercially available. The solution uses delays as a way to allocate the bandwidth. An application with a high delay has a low allocation, and experiences a longer queuing delay for each memory access. In our solution the additional delay is added after the LLC, instead of after the L2, as in the original proposal.

The overhead of prefetch-throttling comes from sampling an application with different prefetcher settings. This is because deactivating prefetching for an application can be detrimental for its performance, especially when prefetching is effective. Likewise, it is detrimental to turn it on prefetching (for a sample period) for an application whose performance is hurt by prefetching.

4 EXPERIMENTAL METHODOLOGY

4.1 Simulated Architecture

We evaluate our proposal on a 16-core tiled CMP architecture modeled using the Sniper Simulator [5]. Each tile has an out-of-order (OOO) core with a private L1 data and instruction cache, a unified private L2 cache and an LLC bank of 512KB. The cache latencies assumed have been modelled using CACTI 6.5 [20]. Details about the baseline architecture are shown in Table 1. A sensitivity study is provided for the CBP parameters in Section 5.2.

Cores	16 cores, x86-64 ISA, 4GHz, OOO, Nehalem-like, 128 ROB entries, dispatch width 4
L1 caches	32KB, 8-way set-associative, split D/I, 1-cycle latency
L2 caches	128KB private per-core, 8-way set-associative, inclusive, 6-cycle data and 2-cycle tag latency
LLC	512KB per-tile, 16-way set-associative, inclusive, 9-cycle data and 2-cycle tag latency, LRU
Coherence protocol	MESIF-protocol, 64 B lines, in-cache directory
Global NoC	4x4 mesh, 4-cycles hop latency (3-cycle pipelined routers, 1-cycle links)
Memory controllers	4 MCUs, 1 channel/MCU, latency 80 ns, 16GB/s per channel
Prefetcher	stride-based, located in L2, 4 prefetches stop at page boundary, 8 flows/core
CBP parameters	*reconfiguration_interval*=10ms *prefetch_sampling_period*=0.5ms, *speedup_threshold* = 1.05, *prefetch_interval*=10ms, *min_bandwidth_allocation*=1, *min_ways*=4

Table 1: Configuration of the simulated 16-core tiled CMP.

4.2 Methodology

We use the entire SPEC CPU2006 suite in our evaluation. The applications are in the format of whole program pinballs [29]. We create

w#	Types	Benchmarks
w1	4CS-BS-PS,5CS-BS,3BS-PS,3CS,1BS	xalancbmk(xa),gromacs(gr),libquantum(li)(2) h264ref(h2),zeusmp(ze),tonto(to),soplex(so), lbm(lb),perlbench(pe),calculix(ca),milc(mi) sphinx3(sp),bwaves(bw),gobmk(go),gamess(ga)
w2	3CS-BS-PS,5CS-BS, 5BS-PS,2CS,1BS	lb,to,pe,go,gcc(gc),mi,li(2),namd(na), h2,cactusADM(cac),ze(2),ca,so,astar(as)
w3	6BS-PS,CS,5BS,4I	bw(2),povray(po)(2),sjeng(2),sp(2),na(2),ze, GemsFDTD(Ge),cac,li,mi,wrf(wr)
w4	CS-BS-PS,2CS-BS,5BS-PS,3CS,2BS,3I	po,bw(2),h2,sjeng(sj),li(2),gr,na,mi(2),as,Ge, ga,wr,lb
w5	5CS-BS-PS,10CS-BS,BS-PS	dealII(de),omnetpp(om)(2),go(2),hmmer(hm),xa leslie3d(le),bzip2(bz)(2),gc,so,mcf(mc),pe,ca(2)
w6	3CS-BS-PS,5CS-BS,4BS-PS,2CS,2BS	sp,bw(2),h2,om,li,gr,go,mi(2),as,hm,ga,le,lb,ca
w7	2CS-BS-PS,2CS-BS,3BS-PS,5CS,4I	po(2),to,sj,h2(2),na,lb(2),ze(2),gr,Ge,as,wr,ga
w8	4CS-BS-PS,4CS-BS,2CS-PS,3BS-PS 3BS	de,bw(3),xa,mi(3),om,li(2),bz,go,so,hm,pe
w9	2CS-BS-PS,5CS-BS,2BS-PS,3CS,BS,2I	gc,po,to,hm,sj,h2,bz,ze,gr,so,Ge,as,pe,wr,ga,cac
w10	2CS-BS-PS,3CS-BS,6BS-PS,CS,2BS,2I	sj,bw(2),de,na,li(2),om,ze,mi(2),xa,Ge,bz,wr,gc
w11	2CS-BS-PS,4CS-BS,4BS-PS,CS,2BS,3I	po,om,sj,go,na(2),le,ze,xa,Ge,bz,wr,ca,sj,sp,gc
w12	6CS-BS-PS,8CS-BS,2CS	de,to,go,h2(2),hm,gr,xa,as(2),bz,ga,gc,lb,so,ca
w13	3CS-BS-PS,2CS-BS,4BS-PS,4CS,3I	to,po,h2,sj,gr,na,as,ze,ga,Ge,lb(2),li,to,mi,wr
w14	5CS-BS-PS,2CS-BS,5BS-PS,CS,BS,2I	de,bw,go,po,hm,na,xa,ze,so,Ge,mc,li, pe,mi,ca,wr

Table 2: 16-core workload.

14 workload mixes (each comprising 16 applications) by randomly selecting applications from the entire SPEC CPU2006 suite. Details about workload mixes are presented in Table 2.

We fast-forward for 16B instructions (in total) and then carry out detailed simulation until all benchmarks have completed at least 500M instructions. Statistics are reported based on the detailed simulation of 500M instructions. After this period the applications continue to run and compete for resources to avoid having a lighter load on long running applications. The methodology is in line with earlier works [3, 19, 28].

4.3 Metrics

We report normalized weighted speedup over baseline for each workload. This is computed by $\frac{1}{N}\sum_{i=1}^{N}\frac{IPC_{i,RM}}{IPC_{i,baseline}}$, in order to evaluate system performance for multi-programmed workloads. RM refers to the system with a resource manager that manages cache, bandwidth and prefetcher settings and the baseline refers to a system with unpartitioned cache and bandwidth and without prefetching.

We also report average normalized turnaround time (ANTT) for each workload since this is a user-oriented performance metric which shows fairness. ANTT is given by $\frac{1}{N}\sum_{i=1}^{N}\frac{CPI_{i,RM}}{CPI_{i,baseline}}$

4.4 Comparison

Table 3 shows the resource managers we evaluate, in addition to CBP, and the corresponding settings they use for cache, bandwidth and prefetching. The baseline configuration represents a system with unpartitioned cache, unpartitioned bandwidth and with prefetching disabled. *equal off* configuration represents a system where cache and bandwidth are equally partitioned and prefetching is disabled. *only cache* represents the configuration where cache is partitioned as described in Section 3.2.1, while bandwidth is unpartitioned and with prefetching disabled. Likewise, *only bw* represents the configuration where bandwidth is partitioned as described in Section 3.2.2 while the cache is unpartitioned and with prefetching disabled. As for *only pref*, prefetch throttling is performed as described in Section 3.2.3 while cache and bandwidth remains unpartitioned. The resource managers that jointly manage two out of the three resources (*bw+perf, bw+cache, cache+perf*) in a coordinated manner can leverage a subset of the interactions described in Section 3.3. We also compare against *CPpf* [33], a recently proposed technique for jointly managing prefetching and cache partitioning. In *CPpf*, prefetch friendly applications are allocated small partition sizes (because the benefit from prefetching can offset the performance drop from small allocation) while the rest of the cache is allocated to the non prefetch friendly applications. In our implementation, we give minimum allocation to the prefetch friendly applications and use UCP (see Section 3.2.1), to partition the remaining capacity among the non prefetch friendly applications. We use UCP and per application partitioning, in order to not put *CPpf* at a disadvantage in comparison to other schemes. Finally, CBP jointly manages all the three techniques dynamically.

RM	cache setting	bandwidth setting	prefetch setting
baseline	unpartitioned	unpartitioned	disabled
equal off	equal	equal	disabled
only cache	dynamic (see 3.2.1)	unpartitioned	disabled
only bw	unpartitioned	dynamic (see 3.2.2)	disabled
only pref	unpartitioned	unpartitioned	dynamic (see 3.2.3)
bw+pref	unpartitioned	dynamic	dynamic
bw+cache	dynamic	dynamic	disabled
cache+pref	dynamic	unpartitioned	dynamic
CPpf	dynamic	unpartitioned	enabled
CBP	dynamic	dynamic	dynamic

Table 3: Configurations evaluated

5 EVALUATION

We first compare CBP to other resource managers that manage a subset of resources. Next, we carry out sensitivity analysis for the different design parameters, to understand its impact on the performance of CBP.

5.1 CBP Performance Analysis

Figure 9 shows normalized weighted speedup for each of the 14 workloads with the bars representing the different resource managers. We use normalized weighted speedup as a measure of performance for the entire workload.*equal off* improves performance in 12 of the mixes and improves performance by 10% on average over the baseline. *only bw* improves performance in 7 of the mixes and on average by 4% over the baseline. *only pref* improves performance

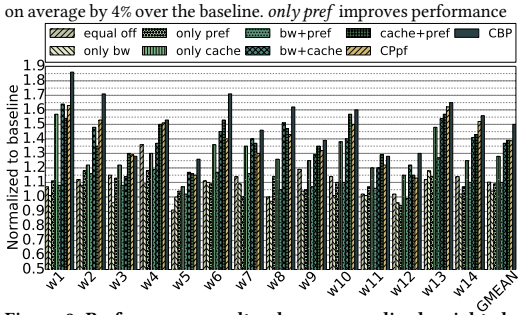


Figure 9: Performance results, shows normalized weighted speedup over baseline.

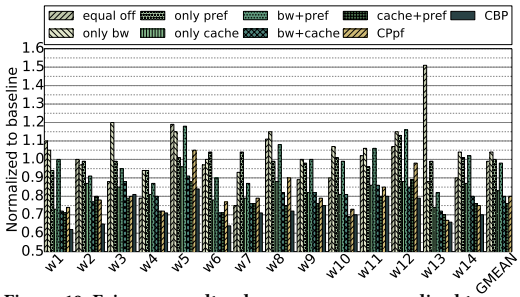


Figure 10: Fairness results, shows average normalized turnaround time (ANTT) over baseline, where lower is better.

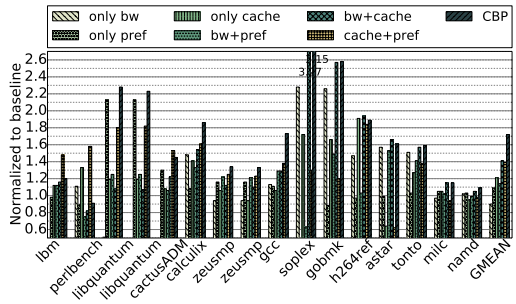


Figure 11: Results for workload 2.

for 12 workloads and provides an average improvement of 9%. *only cache* improves performance for all the workloads and provides an average improvement of around 28%.

Coordinated management of bandwidth partitioning and prefetch throttling (*bw+pref*) leads to higher performance in comparison to the baseline in 12 workloads and an average overall improvement of 10%. Coordinated management of bandwidth and cache partitioning (*cache+bw*) improves performance across all workloads and provides an average performance improvement of 37% (up to 64%). Coordinated cache partitioning and prefetch throttling (*cache+pref*) improves performance across all workloads on average by 39% (up to 57%). *CPpf*, cache partitioning influenced by prefetching, improves performance by 39% (up to 63%).

Among the resource managers that performs coordinated management of two resources, *cache+pref* and *CPpf* achieve the best performance. The results also show that the improvement achieved with coordinated management of two techniques, is larger than summing the improvements from individual techniques. This shows that coordinated management helps exploit synergistic interactions among the different techniques, which cannot otherwise be leveraged. Finally, CBP turns out to be the best performing coordinated resource manager in 14 of the 15 workloads and provides an average improvement of 50% (up to 86%). CBP improves performance by an additional 11% in comparison to the best performing resource manager that does coordinated management of two techniques, as well as state-of-the-art. In one workload, w3, CBP achieves slightly lower performance (2%) in comparison to *cache+pref*. This is because bandwidth partitioning is not very effective for this specific workload.

Figure 10 shows the average normalized turnaround time which shows the fairness of the different resource managers. Note that a lower value signifies greater fairness. On average, CBP shows 27% better fairness than the baseline and 4% better fairness than the best combination of two techniques, *cache+pref*. *cache+pref* has 4% better fairness than *CPpf*.

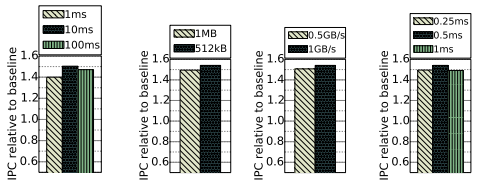
Case study: We investigate the performance of a single workload in detail to understand how CBP improves performance in comparison to resource managers that manage a subset of the techniques. Figure 11 shows the IPC for individual applications in a specific workload (w2) normalized to the baseline IPC. We have classified the applications in this workload into two groups. Group 1 comprises applications, from lbm to gcc (in the figure), for which the *cache+pref* resource manager performs better than the *bw+cache* resource manager. Group 2 comprises the rest of the applications in the workload, from soplex to namd, where the *bw+cache* resource manager performs better than the *cache+pref* resource manager. *cache+pref* resource manager provides the best performance for applications in group 1 because the applications are comparatively more memory intensive and get a larger share of the available bandwidth using this resource manager. Applications in group 2 benefit from bandwidth partitioning since they then get a fair bandwidth share, and in addition are not sensitive to prefetching. When we perform coordinated management of all the three resources, we would ideally prefer to have allocation decisions made by *cache+pref* resource manager for applications in group 1 and *bw+cache* resource manager for applications in group 2. With CBP, some applications in group 1 end up with a lower allocation of bandwidth (compared to *cache+pref*) which hurts their performance (see lbm, perlbench, cactusADM) while the rest of the applications in the group see a performance improvement from getting the right amount of the allocation. For the applications in group 2, CBP manages to match the performance of *bw+cache* resource manager. In summary, CBP enables better trade-offs resulting in a solution that improves overall performance for the workload and outperforms other resource managers that only manage a subset of the techniques.

5.2 Sensitivity analysis

We investigate the sensitivity of CBP to different design parameters in this section.

5.2.1 Impact of reconfiguration interval. The reconfiguration interval, determines how frequently the different resource allocation controllers are invoked when running a workload. We investigate the sensitivity of CBP to different reconfiguration interval values, in order to determine an appropriate interval. Figure 12a shows the average (geo. mean) performance when using three different reconfiguration intervals - 1ms, 10ms and 100ms. A shorter reconfiguration period has the potential to adapt faster to phase change behaviour. However, it also incurs a higher overhead because a larger fraction of the interval would be used up for IPC sampling (required for prefetch throttling). Overall, the results show that using a 10ms period provides a good trade-off between quick adaptation and the overhead incurred for IPC sampling.

5.2.2 Impact of cache size. The results thus far assume that each tile has a baseline cache allocation of 512kb. The total LLC capacity assuming a 16-core tiled CMP would be 8MB. We next study the



(a) Sensitivity for reconfiguration period. **(b) Sensitivity for larger cache size.** **(c) Sensitivity for minimum bandwidth allocation.** **(d) Sensitivity for sample period in prefetch throttling.**

Figure 12: Sensitivity analysis.

impact of changing the cache capacity available for a single tile to 1MB. Figure 12b shows the average performance achieved using CBP with different per-tile capacity normalized to the baseline configuration with the same capacity. The results show that increasing the available LLC capacity leads to a 5% drop in aggregate performance with CBP. This performance drop can be attributed to an increase in cache access time for the larger cache.

5.2.3 Impact of changing bandwidth partitioning parameters. We investigate the sensitivity to the minimum bandwidth allocation, used in the bandwidth allocation algorithm presented in Section 3.2.2. Figure 12c shows the difference in performance with a minimum bandwidth allocation of 0.5GB/s and 1GB/s, normalized to the baseline. There were small variations among the different workloads and some experience a slight improvement while others workloads experience a small drop in performance. Overall, reducing the minimum bandwidth allocation did not have a considerable impact on the performance of CBP as long as it is sufficient for workloads which experience very little queuing delay.

5.2.4 Impact of changing prefetch sampling interval. We finally investigate the impact of changing the *prefetch_sampling_period* used in the prefetch throttling controller 3.2.3. Figure 12d shows the impact of changing the sampling period on performance normalised to the baseline. The intervals we use for evaluation are 0.25ms, 0.5ms and 1ms. The advantage of using a shorter sampling period is that it carries a lower overhead, while the drawback is the risk of over/under estimating the performance benefit from prefetching. The results indicate the sampling interval of 0.5ms achieves the best performance.

6 RELATED WORK

Isolated Management: Several techniques have been proposed in the literature that focus specifically on cache partitioning, bandwidth partitioning and prefetch throttling. Cache partitioning techniques [10, 15, 19, 26, 28] help improve performance and achieve better utilization of available cache resources, by avoiding interference among co-running applications and reducing the number of accesses to memory. Bandwidth partitioning techniques [13, 17, 24, 32], reduce average memory access penalty, by dynamically determining how bandwidth must be shared among the co-running applications. Prefetching can hide memory access latency by fetching the data before it is requested [6, 7, 14, 16, 21, 22]. However, inaccurate prefetches can impact application performance since it can increase the number and cost of demand misses [9]. Prefetch throttling [6, 30], involves adaptively tuning when and what prefetcher settings are used dynamically based on application characteristics and has been shown to provide better performance and address drawbacks of prefetching. The aforementioned works, consider each of the techniques in isolation and leaves room for improvement, as shown in this work, since they do not take the interaction between cache partitioning, bandwidth partitioning and prefetch throttling into account.

Coordinated Management: Several works have proposed combining two of the techniques in order to exploit the benefits from coordination. These works can be broadly classified into the following groups: i) coordinated cache and bandwidth partitioning [23, 27], ii) coordinated prefetching and cache partitioning [31, 33], and iii) coordinated bandwidth partitioning and prefetching [18]. Sahu et al. propose [27] a method for cache and bandwidth partitioning, using a CPI model for bandwidth and set partitioning for the cache. CoPart [23] combines bandwidth and cache partitioning using a user-level run-time. Unlike CBP, their goal is to improve fairness. Recently, CPpf [33] and Sun et.al [31] propose a coordinated approach for cache partitioning and prefetch where prefetch friendly applications where given a smaller cache allocation. Unlike CBP which maintains per-application partitions, cache partitioning in these two proposals is performed for groups of applications. Ebrahimi et al. [8] propose general mechanisms to make memory scheduling techniques prefetch aware. However, these works cannot use the additional interactions and trade-offs which are available when coordinately managing all three resources, which we have shown is important for performance.

Some works have also proposed coordinated management of multiple resources. For instance, CLITE [25] uses bayesian optimization to provide theoretically-grounded resource partitioning to meet QoS targets of multiple resources (e.g., cores, caches, memory bandwidth, memory capacity, disk bandwidth etc.) among multiple co-located jobs. Bitirgen et al. [4] use machine learning to manage power, cache and bandwidth in a coordinated way to anticipate system-level performance impact of allocation decisions. However, in neither of these works is prefetch throttling considered, which we have shown is important in order to realise the full potential of coordinated resource management. To the best of our knowledge, CBP is the first coordinated resource manager for cache partitioning, bandwidth partitioning and prefetch throttling.

7 CONCLUSIONS

We present CBP, a mechanism for coordinated management of cache partitioning, bandwidth partitioning and prefetch throttling. The design is motivated by our in-depth characterisation of the performance impact of cache, bandwidth and prefetch allocation and their interactions. CBP combines local resource allocation controllers with a coordination mechanism that dynamically manages and allocates the resources, in a way which considers both inter- and intra-application interactions. Our evaluation, on a tiled 16-core CMP, demonstrates that CBP improves performance by up to 86% (geo. mean 50%) compared to a system without partitioning and prefetching and by up to 36% (geo. mean 11%) over the state-of-the-art technique that manages cache partitioning and prefetching in a coordinated manner.

REFERENCES

[1] [n.d.]. Intel® 64 and IA-32 Architectures Software Developer Manuals. https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html#nine-volume

[2] [n.d.]. Introduction to Memory Bandwidth Allocation. https://software.intel.com/content/www/us/en/develop/articles/introduction-to-memory-bandwidth-allocation.html

[3] Nathan Beckmann and Daniel Sanchez. 2013. Jigsaw: Scalable software-defined caches. In *Proc. PACT-22*. https://doi.org/10.1109/PACT.2013.6618818

[4] Ramazan Bitirgen, Engin Ipek, and José F. Martínez. 2008. Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach. *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*, 318–329. Issue 2008 PROCEEDINGS. https://doi.org/10.1109/MICRO.2008.4771801

[5] Trevor E. Carlson, Wim Heirman, Stijn Eyerman, Ibrahim Hur, and Lieven Eeckhout. 2014. An Evaluation of High-Level Mechanistic Core Models. *ACM TACO* (2014). https://doi.org/10.1145/2629677

[6] F. Dahlgren, M. Dubois, and P. Stenstrom. 1993. Fixed and Adaptive Sequential Prefetching in Shared Memory Multiprocessors. In *1993 International Conference on Parallel Processing - ICPP'93*, Vol. 1. 56–63. https://doi.org/10.1109/ICPP.1993.92

[7] Fredrik Dahlgren and Per Stenström. 1996. Evaluation of hardware-based stride and sequential prefetching in shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems* 7, 385–398. Issue 4. https://doi.org/10.1109/71.494633

[8] Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N. Patt. 2011. Prefetch-aware shared resource management for multi-core systems. *ACM SIGARCH Computer Architecture News* 39 (2011), 141. Issue 3. https://doi.org/10.1145/2024723.2000081

[9] Eiman Ebrahimi, Onur Mutlu, Chang Joo Lee, and Yale N Patt. 2009. Coordinated control of multiple prefetchers in multi-core systems. 316. https://doi.org/10.1145/1669112.1669154

[10] Nosayba El-Sayed, Anurag Mukkara, Po An Tsai, Harshad Kasture, Xiaosong Ma, and Daniel Sanchez. 2018. KPart: A Hybrid Cache Partitioning-Sharing Technique for Commodity Multicores. *Proceedings - International Symposium on High-Performance Computer Architecture* 2018-Febru, 104–117. https://doi.org/10.1109/HPCA.2018.00019

[11] Babak Falsafi and Thomas F. Wenisch. 2014. A primer on hardware prefetching. *Synthesis Lectures on Computer Architecture* 28 (5 2014), 1–69. https://doi.org/10.2200/S00581ED1V01Y201405CAC028

[12] N. Holtryd, M. Manivannan, P. Stenström, and M. Pericàs. 2020. DELTA: Distributed Locality-Aware Cache Partitioning for Tile-based Chip Multiprocessors. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 578–589. https://doi.org/10.1109/IPDPS47924.2020.00066

[13] D. R. Hower, H. W. Cain, and C. A. Waldspurger. 2017. PABST: Proportionally Allocated Bandwidth at the Source and Target. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 505–516. https://doi.org/10.1109/HPCA.2017.33

[14] Sushant Kondguli and Michael Huang. 2018. Division of labor: A more effective approach to prefetching. *Proceedings - International Symposium on Computer Architecture*, 83–95. https://doi.org/10.1109/ISCA.2018.00018

[15] Hyunjin Lee, Sangyeun Cho, and Bruce R. Childers. 2011. CloudCache: Expanding and shrinking private caches. In *Proc. HPCA-17*. https://doi.org/10.1109/HPCA.2011.5749731

[16] Jaekyu Lee, Hyesoon Kim, and Richard Vuduc. 2012. When Prefetching Works, When It Doesn't, and Why. *ACM Transactions on Architecture and Code Optimization* 9 (3 2012), 1–29. Issue 1. https://doi.org/10.1145/2133382.2133384

[17] F. Liu, X. Jiang, and Y. Solihin. 2010. Understanding how off-chip memory bandwidth partitioning in Chip Multiprocessors affects system performance. In *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*. 1–12. https://doi.org/10.1109/HPCA.2010.5416655

[18] Fang Liu and Yan Solihin. 2011. Studying the impact of hardware prefetching and bandwidth partitioning in chip-multiprocessors. *Proceedings of the ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems - SIGMETRICS '11*, 37. https://doi.org/10.1145/1993744.1993749

[19] R Manikantan, Kaushik Rajan, and R Govindarajan. 2012. Probabilistic shared cache management (PriSM). In *Proc. ISCA-39*. https://doi.org/10.1109/ISCA.2012.6237037

[20] N. Muralimanohar, R. Balasubramonian, and N. Jouppi. 2007. Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0. In *Proc. MICRO-40*. https://doi.org/10.1109/MICRO.2007.33

[21] K.J. Nesbit, A.S. Dhodapkar, and J.E. Smith. 2004. AC/DC: an adaptive data cache prefetcher. *Proceedings. 13th International Conference on Parallel Architecture and Compilation Techniques, 2004. PACT 2004.*, 135–145. https://doi.org/10.1109/pact.2004.1342548

[22] Kyle J. Nesbit and James E. Smith. 2005. Data cache prefetching using a global history buffer. *IEEE Micro* 25 (1 2005), 90–97. Issue 1. https://doi.org/10.1109/MM.2005.6

[23] Jinsu Park, Seongbeom Park, and Woongki Baek. [n.d.]. CoPart: Coordinated Partitioning of Last-Level Cache and Memory Bandwidth for Fairness-Aware Workload Consolidation on Commodity Servers. *Proceedings of the Fourteenth EuroSys Conference 2019* 19. https://doi.org/10.1145/3302424.3303963

[24] Jinsu Park, Seongbeom Park, Myeonggyun Han, Jihoon Hyun, and Woongki Baek. 2018. HyPart: A hybrid technique for practical memory bandwidth partitioning on commodity servers. *Parallel Architectures and Compilation Techniques - Conference Proceedings, PACT*, 1–14. https://doi.org/10.1145/3243176.3243211

[25] T. Patel and D. Tiwari. 2020. CLITE: Efficient and QoS-Aware Co-Location of Multiple Latency-Critical Jobs for Warehouse Scale Computers. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 193–206. https://doi.org/10.1109/HPCA47549.2020.00025

[26] Moinuddin K. Qureshi and Yale N. Patt. 2006. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proc. MICRO-49*. https://doi.org/10.1109/MICRO.2006.49

[27] Aryabartta Sahu and Saparapu Ramakrishna. 2014. Creating heterogeneity at run time by dynamic cache and bandwidth partitioning schemes. *Proceedings of the ACM Symposium on Applied Computing*, 872–879. https://doi.org/10.1145/2554850.2554992

[28] Daniel Sanchez and Christos Kozyrakis. 2011. Vantage: Scalable and Efficient Fine-Grain Cache Partitioning. *Proc. ISCA-38* (2011). https://doi.org/10.1145/2000064.2000073

[29] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. 2002. Automatically Characterizing Large Scale Program Behavior. In *Proc. ASPLOS-10* (San Jose, California). https://doi.org/10.1145/605397.605403

[30] Santhosh Srinath, Onur Mutlu, Hyesoon Kim, and Yale N. Patt. 2007. Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture (HPCA '07)*. IEEE Computer Society, USA, 63–74. https://doi.org/10.1109/HPCA.2007.346185

[31] Gongjin Sun, Junjie Shen, and Alexander V. Veidenbaum. 2019. Combining prefetch control and cache partitioning to improve multicore performance. *Proceedings - 2019 IEEE 33rd International Parallel and Distributed Processing Symposium, IPDPS 2019*, 953–962. https://doi.org/10.1109/IPDPS.2019.00103

[32] Yaocheng Xiang, Chencheng Ye, Xiaolin Wang, Yingwei Luo, and Zhenlin Wang. 2019. EMBA: Efficient Memory Bandwidth Allocation to Improve Performance on Intel Commodity Processor. In *Proceedings of the 48th International Conference on Parallel Processing* (Kyoto, Japan) *(ICPP 2019)*. Association for Computing Machinery, New York, NY, USA, Article 16, 12 pages. https://doi.org/10.1145/3337821.3337863

[33] Jun Xiao, Andy D. Pimentel, and Xu Liu. 2019. CPpf: A prefetch aware LLC partitioning approach. *ACM International Conference Proceeding Series*, 1–10. https://doi.org/10.1145/3337821.3337895